
Article:
Marco, VS, Taylor, B, Wang, Z orcid.org/0000-0001-6157-0662 et al. (1 more author) (2020) Optimizing Deep Learning Inference on Embedded Systems Through Adaptive Model Selection. ACM Transactions on Embedded Computing Systems, 19 (1). 2. ISSN 1539-9087

https://doi.org/10.1145/3371154

Optimizing Deep Learning Inference on Embedded Systems Through Adaptive Model Selection

VICENT SANZ MARCO*, Osaka University, Japan
BEN TAYLOR*, Lancaster University, United Kingdom
ZHENG WANG, University of Leeds, United Kingdom
YEHIA ELKHATIB, Lancaster University, United Kingdom

Deep neural networks (DNNs) are becoming a key enabling technique for many application domains. However, on-device inference on battery-powered, resource-constrained embedding systems is often infeasible due to prohibitively long inferencing time and resource requirements of many DNNs. Offloading computation into the cloud is often unacceptable due to privacy concerns, high latency, or the lack of connectivity. While compression algorithms often succeed in reducing inferencing times, they come at the cost of reduced accuracy.

This paper presents a new, alternative approach to enable efficient execution of DNNs on embedded devices. Our approach dynamically determines which DNN to use for a given input, by considering the desired accuracy and inference time. It employs machine learning to develop a low-cost predictive model to quickly select a pre-trained DNN to use for a given input and the optimization constraint. We achieve this by first off-line training a predictive model, and then using the learned model to select a DNN model to use for new, unseen inputs. We apply our approach to two representative DNN domains: image classification and machine translation. We evaluate our approach on a Jetson TX2 embedded deep learning platform, and consider a range of influential DNN models including convolutional and recurrent neural networks. For image classification, we achieve a 1.8x reduction in inference time with a 7.52% improvement in accuracy, over the most-capable single DNN model. For machine translation, we achieve a 1.34x reduction in inference time over the most-capable single model, with little impact on the quality of translation.

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

ACM Reference Format:
Vicent Sanz Marco, Ben Taylor, Zheng Wang, and Yehia Elkhatib. 2019. Optimizing Deep Learning Inference on Embedded Systems Through Adaptive Model Selection. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2019), 25 pages. https://doi.org/10.1145/3371154

1 INTRODUCTION

Deep learning is getting a lot of attention recently, and with good reason. It has proven ability in solving many difficult problems such as object recognition [13, 28], facial recognition [51, 66], speech processing [2], and machine translation [3]. While many of these tasks are also important application domains for embedded systems [39], existing deep learning solutions are often resource

*Both co-authors contributed equally to this research.

A preliminary version of this article appeared in ACM LCTES 2018 [68].
Authors' addresses: Vicent Sanz Marco, Osaka University, Japan, v.sanzmarco@cmc.osaka-u.ac.jp; Ben Taylor, Lancaster University, United Kingdom, b.d.taylor@lancaster.ac.uk; Zheng Wang, University of Leeds, United Kingdom, z.wang5@leeds.ac.uk; Yehia Elkhatib, Lancaster University, United Kingdom, y.elkhatib@lancaster.ac.uk.

intensive tasks, consuming a considerable amount of CPU, GPU, memory, and power [6]. Without a solution, the hoped-for advances on smart embedded sensing will not be realized.

Numerous optimization tactics have been proposed to enable deep inference[1] on embedded devices. Prior approaches are either architecture specific [64], or come with drawbacks. Model compression is a commonly used technique for accelerating deep neural networks (DNNs). Using compression, a DNN can be optimized by reducing its resource and computational requirements [19, 25, 26, 30]. Unfortunately, this also comes at the cost of a reduction in model accuracy. To avoid this, alternate approaches have been developed; offload some, or all, computation to a cloud server where resources are available for fast inference times [35, 69]. This, however, is not always possible due to high network latency or poor reliability [14]. Furthermore, sending sensitive data over a network could be prohibited due to privacy constraints.

Our work seeks to offer an alternative to execute pre-trained DNN models on embedded systems. Our aim is to design a *generalizable* approach to optimize DNNs to run *efficient* inference without affecting accuracy. Central to our approach is an adaptive scheme for determining, *at runtime*, which of the available DNNs is the best fit for the input and evaluation criterion. Our key insight is that the optimal model – the model which is able to give the correct input in the fastest time – depends on the input data and the evaluation criterion. In fact, by utilizing multiple DNN models we are able to increase accuracy in some cases. In essence, for a simple input – an image taken under good lighting conditions, with a contrasting background; or a short sentence with little punctuation – a simple, fast model would be sufficient; a more complex input would require a more complex model. Similarly, if an accurate output with high confidence is required, a more sophisticated but slower model would have to be employed – otherwise, a simple model would be good enough.

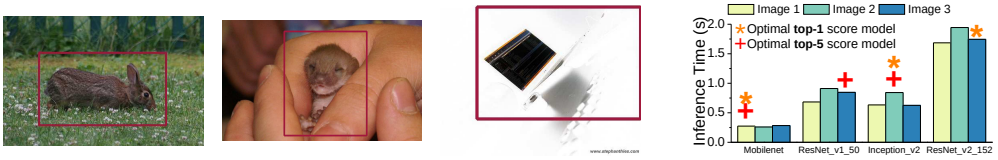
In this work, we employ machine learning (ML) to *automatically* construct a predictor able to dynamically select the optimum model to use. Our predictor is first trained *off-line*. Then, using a set of automatically tuned features of the DNN input, the predictor determines the optimum DNN for a *new*, *unseen* input; taking into consideration the evaluation criterion. We show that our approach can automatically derive high-quality heuristics for different evaluation criteria. The learned strategy can effectively leverage the prediction capability and runtime overhead of candidate DNNs, leading to an overall better accuracy when compared with the most capable DNN model, but with significantly less runtime overhead. Compression can be used in conjunction to our approach to generate multiple DNNs of varying capability, then automatically choose the best at runtime. This is a new way for optimizing deep inference on embedded devices.

Our approach is designed to be generally applicable to all domains of deep learning. As case studies, we choose two typical and unique domains for evaluation: image classification and machine translation. Both domains have a dynamic range of available DNN architectures including convolutional and recurrent neural networks. We evaluate our approach on the NVIDIA Jetson TX2 embedded platform and consider a wide range of influential DNN models, ranging from simple to complex. Experimental results show that our approach delivers portable good performance across the two DNN tasks. For image classification, it improves the inference accuracy by 7.52% over the most-capable single DNN model while achieving 1.8x less inference time. For machine translation, it reduces the inference time of 1.34x over the most-capable model with negligible impact on the quality of the translation.

The paper makes the following technical contributions:

- We present a novel ML based approach to automatically learn how to select DNN models based on the input and precision requirement (Section 3). Our system has little training overhead as it does not require any modification to pre-trained DNN models;

[1]Inference in this work means applying a pre-trained model on an input to obtain the corresponding output.



(a) Image 1 (b) Image 2 (c) Image 3 (d) Inference time

Fig. 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 optimal model (*i.e.* the fastest one that gives an accurate output) depends on the success criterion and the input.

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

	Image 1	Image 2	Image 3
top-5 score	**MobileNet_v1_025**, ResNet_v1_50, Inception_v2, ResNet_v2_152	**Inception_v2**, ResNet_v1_50, ResNet_v2_152	**ResNet_v1_50**, ResNet_v2_152
top-1 score	**MobileNet_v1_025**, ResNet_v1_50, Inception_v2, ResNet_v2_152	**Inception_v2**, ResNet_v2_152	**ResNet_v2_152**

- Our work is the first to leverage multiple DNN models to improve the prediction accuracy and reduce inference time on embedded systems (Section 7).
- Our approach has a good generalization ability as it works effectively on different network architectures, application domains and input datasets. We show that our approach can be easily integrated with existing model compression techniques to improve the overall results.

2 MOTIVATION

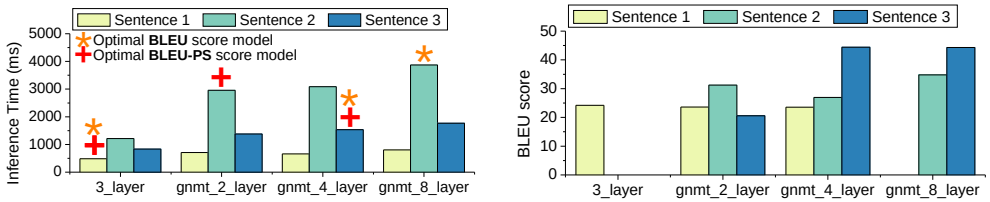
As a motivation, consider two contrasting examples, image classification and machine translation, of using DNNs. The experiments in this section are carried out on a NVIDIA Jetson TX2 platform where we use the GPU for inference; full details of the system can be seen in Section 4.1.

2.1 Image Classification

Setup. For image classification, we investigate one subset of DNNs: Convolutional Neural Networks (CNNs). We compare the performance of three influential CNN architectures: Inception [32], ResNet [29], and MobileNet [30][2]. Specifically, we used the following 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* [58].

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.

[2] Each model architecture follows its own naming convention. MobileNet_v*i*_*j*, where *i* is the version number, and *j* is a width multiplier 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.



(a) Runtime

(b) BLEU scores

Fig. 2. The inference time, optimal model (a), and BLEU score (b) of three sentences (shown in Table 2). Here the optimal model achieves the highest score for an evaluation criteria. Model names explained in footnote 3.

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 Table 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, as the main object is a similar color to the background. In this case the optimal model changes depending on our 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. The results are similar if we use different images of similar complexity levels.

2.2 Machine Translation

Setup. In the second experiment, we consider the following 4 machine translation models as they provide a range of accuracy and runtime capabilities [3]: `3_layer`, `gnmt_2_layer`, `gnmt_4_layer`, and `gnmt_8_layer`. We chose three distinct sentences from the WMT15/16 English-German newstest dataset [71], which can be seen in Table 2.

Evaluation criteria. Unlike image classification, no metrics similar to top-1 and top-5 exist for machine translation. Therefore, we use the following metrics for evaluation:

- ***BLEU*** *(higher is better).* Bilingual Evaluation Understudy is widely used to evaluate machine translation model output. It returns a value between 0 and 1, 1 being a perfect output; it is very rarely achieved.
- ***BLEU-PS*** *(higher is better).* BLEU per second. BLEU is only able to represent a degree of correctness, we also use BLEU-PS to evaluate the trade-off between BLEU and inference time. BLEU-PS is similar to Energy Delay Product (EDP, which is used to evaluate the trade-off between energy consumption and runtime), and is calculated as $\frac{BLEU \times BLEU}{Infer.Time}$.

Results. Figure 2 shows the inference time, BLEU score and optimal model for each sentence. *Sentence 1*, is the simplest sentence, therefore the easiest translation task. The optimal model for all metrics is our simplest, `3_layer`. Surprisingly, our most complex model, `gnmt_8_layer`, fails on this sentence; by using the cheapest model we achieve a higher accuracy 1.66x quicker. Similarly

[3] We name our models using the following convention: `{gnmt_}N_layer`, we prefix the name with `gnmt_`where the model uses the Google Neural Machine Translation Attention [18], and N is the number of layers in the model.

Table 2. The sentences used in Figure 2

Sentence ID	Sentence
1	High on the agenda are plans for greater nuclear co-operation.
2	Advertisements, documentaries, TV series and parts in films consumed his next decade but after his 2008 BBC series, LennyHenry.tv, he thought: " What are you going to do next, Len, because it all feels a bit like you're marking time or you're slightly going sideways."
3	Kenya has started biometrically registering all civil servants in an attempt to remove "ghost workers" from the government's payroll.

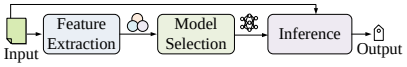


Fig. 3. Overview of our approach.

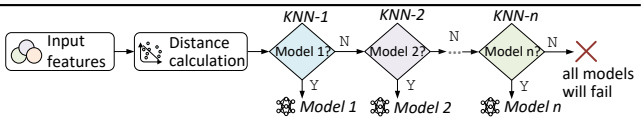


Fig. 4. Our premodel for image classification, made up of a series of KNN models predicting whether to use a specific DNN or not. Our process for selecting classifiers is described in Section 3.3.

the optimal model for *Sentence 3* across both metrics is `gnmt_4_layer`. In this case, we cannot use our cheapest model, as it fails. By choosing the optimal model for *Sentence 3* we can infer 1.15x quicker, without impacting accuracy. It is clear that *Sentence 2* is more complex than *Sentence 1*, it is much longer, has frequent punctuation, and contains non-words, *e.g.* 2008 and TV. In this case, the optimal model changes depending on the evaluation metric. If we are optimising for BLEU-PS we use `gnmt_2_layer`, which is 1.31x times quicker than `gnmt_8_layer`. However, if we would like to maximize accuracy, we need to use `gnmt_8_layer`.

2.3 Summary of Motivation Experiments

The above examples show 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.

3 OUR APPROACH

3.1 Overview

Figure 3 depicts the overall workflow of our approach. Our approach trades memory footprints for accuracy and reduced inference time. At the core of our approach is a predictive model (termed `premodel`) that takes a *new, unseen input* (*e.g.* an image or sentence), and predicts which of a set of pre-trained DNN models to use for that given *input*. This decision may vary depending on the scoring method used at the time, *e.g.* either top-1 or top-5 in image classification.

Our `premodel` is automatically generated based on the problem domain. An example of a generated `premodel` can be seen in Figure 4. The prediction of our `premodel` is based on a set of quantifiable properties – or *features*, such as the number of edges and brightness of an image – of the *input*. Once a model is chosen, the *input* is passed to the selected DNN, which then performs inference on the *input*. Finally, the inference output of the selected DNN is returned. Use of our `premodel` will work in exactly the same way as any single model *i.e.* the input and output will be in the same format, however, we are able to dynamically select the best model to use.

3.2 Premodel Design

To design an effective `premodel` for embedded inference, we consider two design goals: (i) high accuracy and (ii) fast execution time. By correctly choosing the optimal model, a highly accurate `premodel` can reduce the average inference time. Furthermore, a fast `premodel` is important because if a `premodel` takes much longer than any single DNN will be useless. The task of choosing a candidate DNN to use is essentially a classification problem in machine learning. Although using a standard ML classifier as a `premodel` can yield acceptable results, we discovered we can maximize performance by changing the `premodel` architecture depending on the domain (see Section 3.3).

Algorithm 1 Model Selection

```
Require: data, θ, selection_method
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
    improvement_metric = next_selection_metric(selection_method)
    next_DNN = greatest_improvement_DNN(data, curr_DNNs, improvement_metric)
    curr_DNNs.add(next_DNN)
    new_acc = get_acc(curr_DNNs)
    acc_diff = new_acc - curr_acc
    curr_acc = new_acc
end while
```

In this work we consider four well-established classifiers: K-Nearest Neighbour (KNN), a simple clustering based classifier; Decision Tree (DT), a tree based classifier; Naive Bayes (NB), a probabilistic classifier; and Support Vector Machine (SVM), a more complex, but well performing classification algorithm. In Section 7.1, we evaluate a number of different ML techniques, including Decision Trees, Support Vector Machines, and CNNs.

Simultaneously, we consider two different types of premodel architecture: (i) A simple, single classifier architecture using only one ML classifier to predict which DNN to use; (ii) A multiple classifier architecture (See Figure 4), a sequence of ML classifiers where each classifier predicts whether to use a single DNN or not. The later is described in more detail in Section 3.2.1. Finally, we chose a set of features to represent each *input*; the selection process is detailed in in Section 3.5.

3.2.1 Multiple Classifier Architecture. Figure 4 gives an overview of a premodel implementing a multiple classifier architecture. As an example, we will use the KNN based premodel created for image classification. 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 avoid recalculating the distance measurements; instead, 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. 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*, 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.

3.3 Inference Model Selection

In Algorithm 1 we describe our selection process for choosing which DNNs to include in our premodel. This algorithm takes in three parameters: (1) *data*, containing the output of each DNN for every *input*; (2) θ, a threshold parameter telling us when to terminate the model selection process; and (3) *selection_method*, one of a choice of methods that produces an *improvement_metric* (accuracy or optimal) for determining when if a candiate DNN should be included in the premodel in each iteration. We consider the following three model selection methods:

- ***Based on accuracy.*** Using this selection method, we will add a DNN to premodel if it has the greatest improvement in accuracy for each iteration. There are some cases where the selected DNNs all fail to make a correct prediction, but some of the remaining candidate models can. During each selection iteration, we will choose a remaining DNN that if it is included, it can lead to the most significant improvement in prediction accuracy for premodel.

- ***Based on optimal.*** In each iteration of the loop, the most optimal DNN is selected; *i.e.* the one that gives the greatest overall improvement in accuracy, but leads to the lowest increase in inference time, for the selected DNN set.
- ***Alternate.*** A hybrid of the first two approaches. We alternate between choosing the most optimal and the most accurate DNN in each iteration. Our first DNN is always the most optimal.

Model selection process. The model selection process works as follows.

- ***Initialization.*** The first DNN we include is the most optimal model for our training data, i.e., the DNN that is most frequently considered to be optimal across training instances.
- ***Iterative selection.*** At each iteration, we consider each of the remaining potential DNNs, and add the one which brings the greatest improvement to our *improvement_metric* (accuracy or optimal), which can change per iteration based on the *selection_method*.
- ***Termination.*** We iteratively add new DNNs until our accuracy improvement is lower than the termination threshold, θ%.

Using this Model Selection Algorithm we are able to add DNNs that best compliment one another when working together, maximizing our accuracy while keeping our runtime low. In Section 7.2 we evaluate the impact of different parameter choices on our algorithm.

Illustrative example. We now walk through the Model Selection Algorithm using the image classification problem as an example. In this example, we set our threshold θ to 0.5, which is empirically decided through our pilot experiments. We also set *selection_method* to "based on accuracy" for this example. *We carry out a a sensitivity analysis for these parameters later in Section 7.2.* Figure 5 shows the percentage of our training data that considers each of our CNNs to be optimal. For this example, the model selection process works as follows:

- ***First model.*** The first model is the most optimal model. In this example, `MobileNet_v1_100` is chosen to be *Model-1* because it is optimal for most (70.75%) of our training data.
- ***Iterative selection.*** If we were to follow the "based on optimal" selection method and 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 and consider which of our remaining CNNs gives the greatest improvement in accuracy (i.e., "based on accuracy"), as '*Accuracy*' is our *improvement_metric*. Intuitively, as image classification is either right or wrong, we are searching for the CNN that is able to correctly classify the most of the remaining 29.25% cases where `MobileNet_v1_100` fails. As seen in Figure 7b, `Inception_v4` is best, correctly classifying 43.91% of the remaining data and creating a 12.84% increase in `premodel` accuracy. Repeating this process (Figure 7c), we add `ResNet_v1_152` to our `premodel`, further increasing total accuracy by 2.55%.
- ***Termination.*** After adding `ResNet_v1_152`, we iterate once more to achieve a `premodel` accuracy increase of less than 0.5% (θ), and therefore terminate.
- ***Results.*** After running this process, our `premodel` is composed of: `MobileNet_v1_100` for *Model-1*, `Inception_v4` for *Model-2*, and `ResNet_v1_152` for *Model-3*.

3.4 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 6. Generally, we need to figure out which candidate DNN is optimum for each of our training *inputs* (to be used by the Model Selection Algorithm described in Section 3.3), we then train our `premodel` to predict the same for any *new, unseen inputs*.

Generate training data. Our training dataset consists of the feature values and the corresponding optimum DNN for each *input* under an evaluation criterion. To evaluate the performance of the

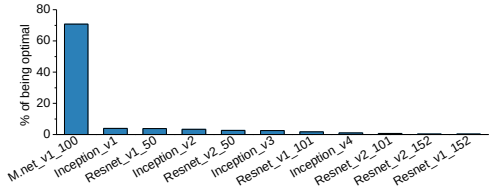


Fig. 5. How often a CNN model is considered to be optimal under top-1 on the training dataset.

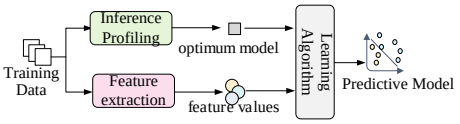
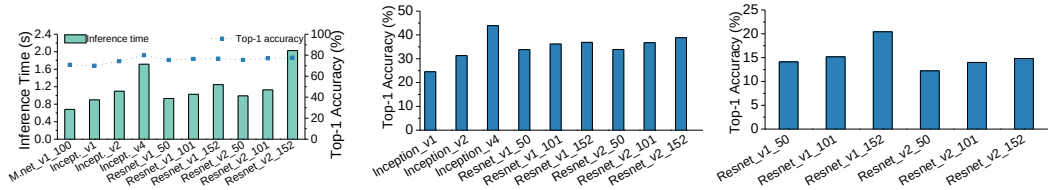


Fig. 6. The training process. We use the same procedure to train each individual model within the premodel for each evaluation criterion.



(a) All CNNs (b) Where MobileNet fails (c) Where Mobilnet & Inception fails.

Fig. 7. Image classification results. (a) The top-1 accuracy and inference time of all CNNs we consider. (b) The top-1 accuracy of all CNNs on the images on which MobileNet_v1_100 fails. (c) The top-1 accuracy of all CNNs on the images on which MobileNet_v1_100 and Inception_v4 fails.

candidate DNN models, they must be applied to unseen *inputs*. We exhaustively execute each candidate DNN on the *inputs*, 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. It is to note that adding a new DNN simply requires executing all *inputs* on the new DNN while taking the same measurements described above.

Using the execution time, and DNN output results, we can calculate the *optimum* classifier for each *input*; *i.e.* the model that achieves the accuracy goal (top-1, top-5, or BLEU) in the least amount of time. Finally, we extract the feature values (described in Section 3.5) from each *input*, and pair the feature values to the optimum classifier for each *input*, resulting in our complete training dataset.

Building the premodel. The training data is used to determine the classification models to use and their optimal hyper-parameters. All classifiers we consider for premodel support are supervized learning algorithms. Therefore, we simply supply the classifier with the training data and it carries out its internal algorithm. For example, 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 candidate inference models to be included. In our case, we had an unusually long training time as we considered a large number of DNN models. We would expect in deployment that the user has a much smaller search space for potential DNNs. The time in model selection and parameter tuning is negligible (less than 2 hours) in comparison. See also Section 7.4.

3.5 Features

One key aspect in building a successful predictor is selecting the right features to characterize the input. In this work, we have developed an automatic feature selection process, the user is simply required to provide a number of candidate features. Automatic feature generation could be used to provide candidate features, however this is out of the scope of this work.

3.5.1 Feature Selection. Feature extraction is the biggest overhead of our premodel, therefore by reducing our feature count we can decrease the total execution time. Moreover, by reducing the number of features we are also improving the generalizability of our premodel.

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 perform 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.

Next, we evaluated the importance of each of our remaining features. To do so, we first trained and evaluated our premodel using K-Fold cross validation (see also Section 7.4) 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 important, otherwise, the feature does not hold much importance for our purposes. Using this information we performed a greedy search, removing the least important features one by one. We detail the outcome of this process in Section 7.3. Below we have summarized the result of each feature selection stage on both of our case studies. Removing any of the remaining features resulted in a significant drop in model accuracy.

3.5.2 Feature Scaling. The final step before passing our features to a ML model is scaling each feature to a common range (between 0 and 1) in order to prevent the range of any single feature being a factor in its importance. Scaling does not affect the distribution or variance of feature values. To achieve this during deployment, we record the minimum and maximum values of each feature in the training dataset and use these to scale the corresponding features of new data.

3.6 Runtime Deployment

Deployment of our proposed method is designed to be simple and easy to use, similar to current DNN usage techniques. We have encapsulated all of the inner workings, such as needing to read the output of the premodel and then choosing the correct DNN model. A user would interact with our premodel by simply calling a prediction function and getting a result in return in the same format as the DNNs in use. Using image classification as an example, the return value would be the predicted labels and their confidence levels.

4 EVALUATION SETUP

We apply our approach to two representative DNN domains: image classification and machine translation. Each domain is presented as a case study (Sections 5 and 6) that shows the results at each stage of applying our approach; providing an end-to-end analysis. The case studies will end with an analysis in Section 7 of how our approach performed against other representative DNNs in the domain. In the remainder of this section, we describe our evaluation setup and methodology.

4.1 Hardware and Software

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.

4.2 Evaluation Methodology

4.2.1 Model Evaluation. We use *10-fold cross-validation* to evaluate each `premodel` on its respective dataset. Specifically, we split our dataset into 10 sets which equally represent the full dataset, *e.g.* if we consider image classification, 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 when the system is idle.
- ***Accuracy*** *(higher is better).* The ratio of correctly labeled cases to the total number of testing cases.

Metrics for image classification. The following metrics are specific to image classification:

- ***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 $2 \times \frac{Recall \times Precision}{Recall + Precision}$. It is useful when the test datasets have an uneven distribution of classes.

Metrics for machine translation. The following metrics are specific to machine translation:

- ***BLEU*** *(higher is better).* Similar to precision in image classification. It is a measure of how much the words (and/or n-grams) in the `DNN` model output appeared in the reference output(s).
- ***Rouge*** *(higher is better).* Similar to recall in image classification. It is a measure of how much the words (and/or n-grams) in the reference output(s) appear in the `DNN` model output.
- ***F1 measure*** *(higher is better).* Similar to F1 score for image classification. The weighted average of BLEU and Rouge, calculated as $2 \times \frac{Rouge \times BLEU}{Rouge + BLEU}$.

4.2.2 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 `DNN` 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 onboard energy sensors at a frequency of 1,000 samples per second. It is to note that our work does not directly optimize for energy consumption. We found that in our scenario there is little difference when optimizing for energy consumption compared to time.

5 CASE STUDY 1: IMAGE CLASSIFICATION

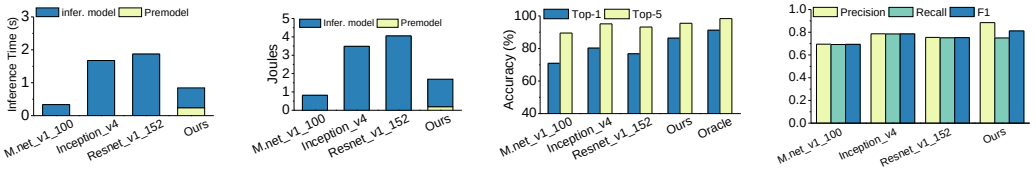
To evaluate our approach in the domain of image classification we consider 14 pre-trained `CNN` models from the TensorFlow-Slim library [63]. The models are built using TensorFlow and trained on the ImageNet ILSVRC 2012 *training set.* We use the Imagenet ILSVRC 2012 *validation set* to create the training data for our `premodel`, and evaluate it using *cross-validation* (see Section 4.2).

Table 3. All features considered for image classification.

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

Table 4. Correlation values (absolute) of removed features to the kept ones for image classification.

Kept Feature	Removed Feature	Correl.
avg_perc_brightness	*perc_brightness_rms*	0.98
	avg_brightness	0.91
	brightness_rms	0.88
edge_length1	*edge_length {4-7}*	0.78 - 0.85
hue1	*hue {2-6}*	0.99



(a) Inference Time (b) Energy Consumption (c) Accuracy (d) Precision, Recall & F1

Fig. 8. Image Classification – Overall performance of our approach against individual models and an `Oracle` for inference time (a), energy consumption (b), accuracy (c), and precision, recall and F1 scores (d).

5.1 Premodel for Image Classification

5.1.1 Feature Selection. In this work, we considered a total of 29 candidate features, shown in Table 3. The features were chosen based on previous image classification work [27], *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). Table 4 summarizes the features removed using correlation-based feature selection, leaving 17 features. Next, we iteratively evaluated feature importance and performed a greedy search that reduced our feature count down to 7 features (see Table 5). This process is described in Section 3.5.1.

5.1.2 Feature Analysis. We now analyze the importance of each feature that was chosen during our feature selection process. We calculate feature importance by first training a `premodel` using all of our chosen features (n), and note the accuracy of our `premodel`. In turn, we then remove each feature, retraining and evaluating our `premodel` on the remaining $n-1$ features, noting the drop in accuracy. We then normalize the values to produce a percentage of importance for each feature. Figure 9a shows the top 5 dominant features based on their impact on our `premodel` accuracy. 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.

5.1.3 Creating The Premodel. Applying our automatic approach to `premodel` creation, described in Section 3.2, resulted in implementing a multiple classifier architecture consisting of a series of simple `KNN` models. We found that `KNN` has a quick prediction time (<1ms) and achieves a high accuracy for this problem. Furthermore, we applied our Model Selection Algorithm (Section 3.3) to determine which `CNNs` to be included in the `premodel`. As we have explained in Section 3.3, this process resulted in a choice of: `MobileNet_v1_100` for *Model-1*, `Inception_v4` for *Model-2*, and, finally, `ResNet_v1_152` for *Model-3*. Finally, we use the training data generated in Section 5.1.1 and 10-fold-cross-validation to train and evaluate our `premodel`.

Table 5. Image Classification – The final chosen features after feature selection.

n_keypoints	*avg_perc_brightness*	*hue1*
contrast	*area_by_perim*	*edge_length1*
aspect_ratio		

Table 6. Machine Translation – The final chosen features after feature selection.

n_words
avg_adj
BoW

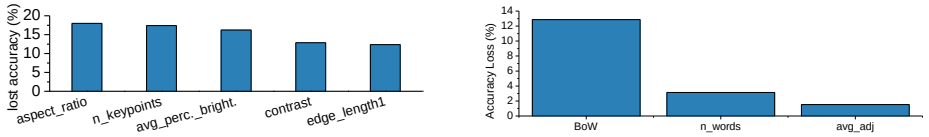
5.2 Overall Performance of Image Classification

5.2.1 Inference Time. Figure 8a compares the inference time among DNN models used by our premodel and our approach. Due to space limitations we limit to these three models (MobileNet_v1_100, Inception_v4, and ResNet_v1_152) since they are the ones used by our premodel. MobileNet_v1_100 is the fastest model for inferencing, being 2.8x and 2x faster than Inception_v4 and ResNet_v1_152, respectively, but is least accurate (see Figure 8c). The average inference time of our approach is under a second, which is slightly longer than the 0.4 second average time of MobileNet_v1_100. Our slower time is a result of using a premodel, and choosing Inception_v4 or ResNet_v1_152 on occasion. Most of the overhead of our premodel comes from feature extraction. Our approach is 1.8x faster than Inception_v4, the most accurate inference model in our model set. Given that our approach can significantly improve the prediction accuracy of MobileNet_v1_100, we believe the modest cost of our premodel is acceptable.

5.2.2 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_v4 and ResNet_v1_152. The energy footprint of our premodel is small, being 4x and 24x lower than MobileNet_v1_100 and ResNet_v1_152 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 infers 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.

5.2.3 Accuracy. Figure 8c compares the top-1 and top-5 accuracy achieved by each approach. We also show the best possible 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_v1_152 will successfully classify some images which Inception_v4 will fail on. Therefore, by effectively leveraging multiple models, our approach outperforms all individual inference models. It improves the accuracy of MobileNet_v1_100 by 16.6% and 6% for the top-1 and the top-5 scores, respectively. It also improves the top-1 accuracy of ResNet_v1_152 and Inception_v4 by 10.7% and 7.6%, respectively. While we observe little improvement for the top-5 score over Inception_v4 – just 0.34% – our approach is 2x faster than it. Our approach delivers over 96% of the Oracle performance (86.3% vs 91.2% for top-1 and 95.4% vs 98.3% for top-5). This shows that our approach can improve the inference accuracy of individual models. Overall, we achieve a 7.52% improvement in accuracy over the most-capable single DNN model, while reducing inference time by 1.8x.

5.2.4 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.



(a) Image Classification (b) Machine Translation

Fig. 9. The loss in accuracy when final chosen features are not used in our premodel. For image classification (a) we only show the top five. For machine translation (b) we show all 3.

Table 7. All features considered for machine translation. See Section 3.5

Feature	Description
n_words	# words in the sentence
n_bpe_chars	# bpe characters in a sentence
avg_bpe	Average number of bpe characters per word
n_tokens	# tokens in the sentence when tokenized
avg_noun	Average number of nouns per word
avg_verb	Average number of verbs per word
avg_adj	Average number of adjectives per word
avg_sat_adj	Average number of satellite adjectives per word
avg_adverb	Average number of adverbs per word
avg_punc	Average punctuation characters per word
avg_word_length	Average number of characters per word

Table 8. Correlation values (absolute) of removed features to the kept ones for machine translation.

Kept Feature	Removed Feature	Correl.
n_words	*n_bpe_chars*	0.96
	n_tokens	0.99

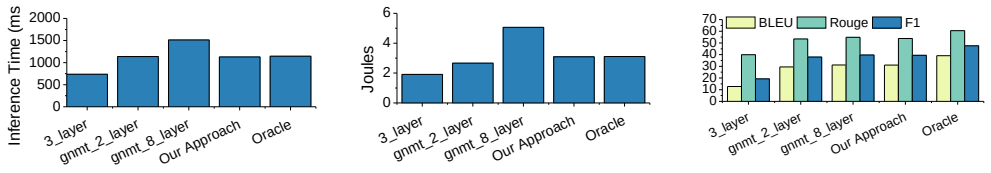
6 CASE STUDY 2: MACHINE TRANSLATION

To evaluate our approach for machine translation we consider 15 DNN models. We include models of varying sizes and architectures, all trained using Tensorflow-NMT, a Neural Machine Translation library provided by Tensorflow [43]. We name our models using the following convention: {gnmt_}N_layer, we prefix the name with gnmt_where the model uses the Google Neural Machine Translation Attention [18], and N is the number of layers in the model. *e.g.* 4_layer is a default Tensorflow-NMT model made up of 4 layers. The models were trained on the *WMT09-WMT14* English-German newstest dataset, and we use the *WMT15/16* English-German newstest dataset [71] to create our premodel training data. Using *10-fold-cross-validation* on our premodel to give a end-to-end analysis of our approach.

6.1 Premodel for Machine Translation

6.1.1 Feature Selection. We considered a total of 11 features, which can be seen in Table 7, and a Bag of Words (BoW) representation of each sentence (explained in more detail below). Similar to image classification, we chose our candidate features based on previous work [36, 42], *e.g.* BoW, as well as intuition based on our motivation (Section 2.1), *e.g. n_words* (longer sentences are more complex and require a more complex translator).

Bag of words. Applying our method to machine translation brings with it the need to classify each sentence to predict the optimal DNN. Text classification is a notoriously difficult task, and is made more difficult when we only have a single sentence to gather features from. We are able to create a successful premodel only using the features described in Table 7. However, with the addition of a Bag of Words (BoW) representation of each sentence we saw an increase in accuracy. Furthermore, previous work in sentence classification [36, 42, 44] often use a BoW representation, suggesting that BoW can be useful for characterizing and modeling a sentence. A BoW representation of text describes the occurrence of words within the text. It is represented as a vector that is based on a vocabulary. We generated a domain specific vocabulary based on all words in our training dataset. Finally, we used Chi-square (Chi2) to perform feature reduction, which is widely used for BoW, leaving us with a BoW feature vector of length 1500. We include a full evaluation of the effect of BoW and Chi2 feature selection on our machine translation premodel in Section 7.3.2.



(a) Inference Time (b) Energy Consumption (c) BLEU, Rouge, and F1

Fig. 10. Machine Translation – Overall performance of our approach against individual models and an `Oracle`.

Table 8 summarizes the features we removed during the first stage of feature selection, leaving 9 features. During the second stage we reduced our feature count down to 3 features (see Table 6). Figure 9b summarizes the accuracy loss by removing any of the three selected features; the two shown in Table 6, and a BoW representation. It can be seen that by including BoW we reach a much higher accuracy. This is to be expected, as BoW is a well researched and used representation of text input. If we remove either *n_words* or *avg_adj* there is a small drop in accuracy, this indicates that BoW is able to capture similar information. We chose to keep both of these features as they bring a small increase to accuracy with negligible overhead.

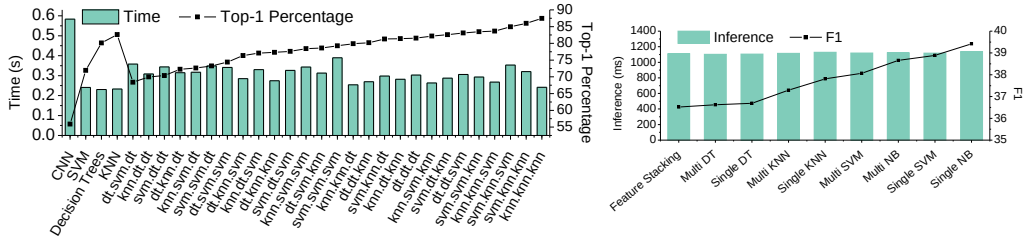
6.1.2 Creating The Premodel. Using our approach resulted in implementing a single `NB` classifier `premodel`. We believe that a single architecture `premodel` was chosen because of our reduced dataset, *i.e.* we have one tenth of the training data compared to image classification. `NB` achieved a high accuracy for this task, and has a quick prediction time (<1ms).

Applying our Model Selection Algorithm, we set *selection_method* to '*Accuracy*' and θ to 2.0. Again, see Section 7.2 for a sensitivity analysis of these parameters. This resulted in a `premodel` selection of `gnmt_2_layer`, `gnmt_8_layer`, and `gnmt_3_layer` for *Model-1*, *Model-2*, and *Model-3*, respectively. Finally, we use the training data generated in Section 6.1.1 and 10-fold-cross-validation to train and evaluate our `premodel`.

6.2 Overall Performance for Machine Translation

In this section, we evaluate our methodology when applied to Neural Machine Translation (`NMT`). We compare our approach to three other `NMT` models considered in our `premodel`. We chose these models as they show a range of complexity and capability. Furthermore, we compare our approach to an `Oracle`, a *theoretical* perfect approach that achieves the best possible score for each metric.

6.2.1 Inference Time. As depicted in Figure 10a, `3_layer` is the quickest `DNN`, 1.55x faster than the `Oracle` and 2.05x faster than the most complex individual `DNN`, `gnmt_8_layer`. However, `3_layer` is also the least accurate `DNN` (Figure 10c) as it is outperformed in every accuracy metric by all other approaches. Our approach, the `Oracle`, and `gnmt_2_layer` have very similar inference times; nonetheless, our approach and the `Oracle` outperform `gnmt_2_layer` for accuracy. The runtime of our `premodel` and feature extraction is small, consisting of <1ms for the `premodel` and <5ms for feature extraction, per sentence. Feature extraction and `premodel` overheads are included in the inference time of our approach and the `Oracle`. Incidentally, our approach is slightly quicker than the `Oracle`; this is a result of our `premodel` often mispredicting `gnmt_2_layer` for `gnmt_8_layer` and vice versa. This specific misprediction makes up 38.5% of the cases where `premodel` makes an incorrect prediction. To improve accuracy we will need more data to train our `premodel`, as we currently have a high feature to sentence ratio. Alternatively, we could deeply investigate the sentences that are best for each model and intuitively add a new feature to our `premodel`, however, the differences may not be intuitive to spot. Overall, we are 1.34x faster than the single most capable `DNN` without a decrease in accuracy.



(a) Image Classification (b) Machine Translation

Fig. 11. Comparison of alternative predictive modeling techniques for building the premodel.

6.2.2 Energy Consumption. Figure 10b compares energy consumption, including premodel costs, which are negligible (See Section 7.4). Much like the image classification DNNs, energy consumption is proportional to model inference time; therefore, as we reduce overall inference time we also improve energy efficiency. A major difference between energy consumption and inference time is the emphasized ratios between each model, *e.g.* gnmt_2_layer is 1.24x quicker than gnmt_8_layer, but it uses 1.90x less energy, nearly half as much. Overall, we use 1.39x less energy on average than the single most capable model, without a significant change in F1 measure. Therefore, our methodology can be used to improve energy efficiency while having little impact on accuracy, or in some cases, seeing an improvement in accuracy. Furthermore, our premodel is able to predict when none of the DNNs are able to give a suitable output, in this case we can skip inference to avoid wasting power. Implementing this results in using 1.48x less energy on average than gnmt_8_layer.

6.2.3 BLEU, Rouge, and F1 Measure. Figure 10c compares DNNs across our accuracy metrics. We will mostly compare F1 measure here, but all metrics follow the same pattern. As all models do not fail on the same sentences, we are able to achieve an overall better F1 measure by leveraging multiple DNNs. This can be seen by looking at the Oracle, which achieves an F1 measure of 47.54, a 20% increase over gnmt_8_layer, which achieves 39.71. For this case study, we achieved 83% of the Oracle F1 measure. Overall our approach achieves approximately the same F1 measure as the single most capable model, and improves upon the accuracy of gnmt_2_layer (the closest single DNN in terms of inference time), by 4%. For our premodel to achieve its full potential, as show by the Oracle, we require more data to train and test our premodel.

7 ANALYSIS

We now analyze the working mechanism of our approach to justify our design choices.

7.1 Alternative Techniques for Premodel

7.1.1 Image Classification. Figure 11a 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 our multi-classifier architecture made up of only KNN classifiers, we have considered different variations of Decision Trees (DT) and Support Vector Machines (SVM). We also consider a single architecture premodel using the above mentioned ML techniques, and a CNN. Our CNN-based premodel is based on the MobileNet structure, which is designed for embedded inference. We train all models using the same examples. We also use the same feature set for KNN, DT, and SVM. For the CNN, we use an automated hyper-parameter tuner [38] to optimize the training parameters, and we train the model for over 500 epochs.

Notation. In this instance, our multiple classifier architecture requires 3 components. We denote a premodel configuration as $X.Y.Z$ (see also Section 3.2.1), where X, Y and Z indicate classifier for the first, second and third level of the premodel, respectively. For example, KNN.SVM.KNN denotes using a KNN model for the first and last levels, with a SVM model at the second level.

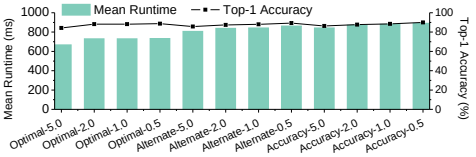


Fig. 12. The inference time and Top-1 accuracy achieved when building a `premodel` based on the Model Selection Algorithm configurations shown.

While we hypothesized a CNN model to be effective, the results are disappointing given its high runtime overhead. A KNN model has an overhead that is comparable to the DT and the SVM, but has a higher accuracy. It is clear that our chosen `premodel` architecture (KNN.KNN.KNN) was the best choice, it achieves 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 neighbouring 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.

7.1.2 Machine Translation. Figure 11b shows the F1 measure and inference time for different architectures of `premodel` when applied to the machine translation problem. In this instance, we are predicting whether to use `gnmt_2_layer`, `gnmt_8_layer`, or `gnmt_3_layer` for translating an input sentence. Our `premodel` can also predict that all these translators will fail, making a total of 4 labels to choose from. We evaluated single and multiple architectures, across KNN, DT, SVM, and NB classifiers. For multiple classifier architectures we carried out a less exhaustive search compared to Section 7.1.1; we discovered that best performance was often achieved by using the same classifier for each component. Finally, we compare an alternate approach named feature stacking [42]. Using feature stacking we split classification into two classifiers, one using the BoW features, the other using our remaining features, we then use a probability measure choose the predicted model.

For this problem we can see that the single classifier architecture always outperforms its multiple classifier alternative. This is likely as a result of our high dimensional feature space, with a comparatively low training set. Feature stacking also had a poor performance for this problem, in fact it performs worse than all other architectures, indicating that our features work better together. Overall, there is little variance in the runtime of each approach, every model achieves a runtime between 1100ms and 1140 ms. Our chosen approach, a single NB classifier, achieves the highest F1 measure overall, with very similar runtime to all other approaches.

7.2 Sensitivity Analysis for Model Selection Algorithm

In Section 3.3, we describe the algorithm we created to decide which DNNs to include in our `premodel`. In this section we will analyze how changing the parameters given to the Model Selection Algorithm effect our `premodel`, and the resultant end-to-end performance. We will perform a case study using image classification, but the results for machine translation are very similar. We consider the performance if we were able to create a perfect predictor as a `premodel`, this is to prevent our `premodel` accuracy from introducing any noise and allowing us to evaluate the Model Selection Algorithm in isolation. a total of 12 parameter configurations – our three available choices for *SelectionMethod* (defined in Section 3.3), and 4 different choices for θ (5.0, 2.0, 1.0, and 0.5). We take every combination of these parameters.

Notation. Our parameter configuration is *SelectionMethod-*θ, where *SelectionMethod* is either *Accuracy*, *Optimal*, or *Alternate*; and θ is our threshold parameter. For example, the notation *Accuracy-5.0*, means we always select the most accurate model in each iteration of our algorithm, and we stop once our accuracy improvement is less than 5.0%.

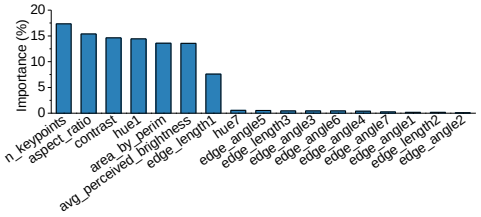


Fig. 13. Image Classification – Accuracy loss if a feature is not used.

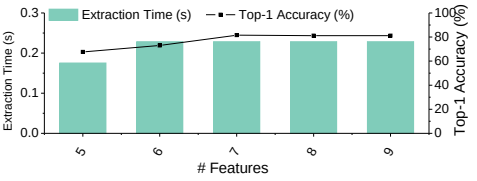


Fig. 14. Image Classification – The impact of `premodel` feature count on `premodel` runtime and overall top-1 score.

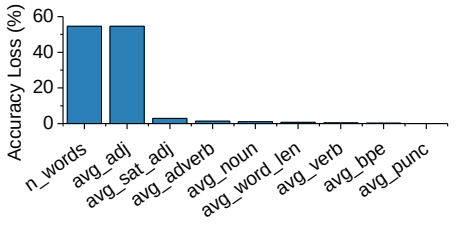


Fig. 15. Machine Translation – Accuracy loss if a certain feature is not used.

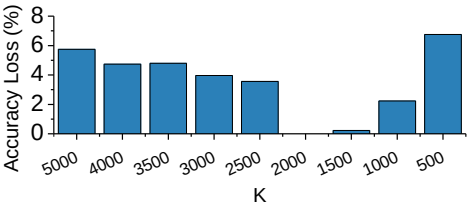


Fig. 16. Machine Translation – Accuracy loss for different values of k using Bag of Words. k=2000 is our baseline.

7.2.1 Results. Figure 12 shows the effect of different parameters on our final `premodel` results. As we decrease θ, our Model Selection Algorithm will select more models to include in our `premodel`, *e.g.* The `premodel` of *Alternate-5.0* to *Alternate-0.5* is made up of 3, 4, 5, and 7 `DNN` classifiers, respectively. Including more `DNNs` results in a higher overall top-1 accuracy, however there are also some drawbacks. More `DNNs` means more classes for our `premodel` to choose between, therefore making the job of the `premodel` harder. It also means that we need to hold more `DNNs` in memory, which could be an issue for devices with limited memory (We discuss resource usage in more detail in Section 7.6.2). It is worth noting that there is no change in `DNN` selection from *Optimal-2.0* to *Optimal-1.0*, as the next model that we can add only brings an accuracy improvement of 0.488.

Finally, we can see that each *SelectionMethod* has its own 'profile', that is, each has its own positive and negative impact. Figure 12 shows that *Optimal* results in an overall faster runtime, however, it has a lower top-1 accuracy. *Accuracy* is able to achieve the highest possible top-1 score, but this comes at the cost of speed, achieving 1.26x slowdown for a 2% accuracy increase. *Alternate* attempts to find a balance between the other two approaches, it is able to achieve and accuracy and runtime in between *Optimal* and *Accuracy*.

7.3 Feature Importance

7.3.1 Image Classification. Our feature selection process (described in Section 3.5) resulted in using 7 features to represent each image to our `premodel`. In Figure 13 we show the importance of all of the chosen features along with other considered ones (given in Tables 3 and 4). The first 7 chosen features are the most important; there is a sudden drop in feature importance at feature 8 (*hue7*). Furthermore, Figure 14 shows the impact on `premodel` execution time and top-1 accuracy when we change the number of features used. 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.

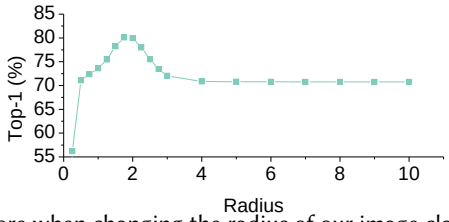


Fig. 17. The top-1 score when changing the radius of our image classification premodel.

7.3.2 Machine Translation. As with image classification above, in this section we will evaluate our feature selection process on the machine translation problem. We will evaluate our BoW feature separately so clearly show the importance of all feature choices. Figure 15 shows the importance of all our features which were not removed during our correlation check (See Table 8). As we discussed in Section 5.1.2, *n_words* and *avg_adj* are essential to premodel accuracy, it is clear that removing either of them severely deteriorates our premodel. If we were to keep *avg_sat_adj*, we would see a 2.9% increase in premodel accuracy, however we choose to leave this out as it provides negligible improvements in the presence of BoW.

Bag of words. We found that including BoW as a feature in our premodel brought improvements in accuracy for little overhead (See Figure 9b). We use the chi-squared test to evaluate each row of our BoW vector, and choose the top k features. In Figure 16 we show the accuracy loss by choosing different values of k, as a baseline we use our chosen value k=2000. It is clear that choosing a value greater than 2000 results in a dramatic loss in accuracy (nearly 4%), which quickly increases as k increases. Setting k=1500 results in a small loss in accuracy, but reducing it further leads to much bigger losses in accuracy. *e.g.* k=500 results in a 5.75% loss in accuracy. This indicates that k>2000 results in our premodel that is prone to overfitting, while k<1500 is unable to capture all of the information required for accurate predictions, therefore the optimal value of k sits around the 2000-1500 mark. We chose k=2000 as we achieved the highest accuracy with this value, and the overhead of increasing k is negligible.

7.4 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 3.4). We can speed this up by using multiple machines. Compared to the training time of a typical DNN, our training overhead is negligible. Because our approach trades RAM space for improved accuracy and reduced inference time, we provide an evaluation of resource utilization in Section 7.6.2. In addition to our case studies, we have evaluated our premodel overhead for object detection, using the COCO dataset [41], where the runtime overhead is similar to image classification, under 13.5%.

Image classification. The runtime overhead of our premodel is minimal, as depicted in Figure 8a. Out of a total average execution time of <1 second to classify an image, our premodel accounts for 28%. In comparison, this is 12.9% and 71.7% of the average execution time of the most (ResNet_v1_152) and least (MobileNet_v1_100) expensive models, respectively. Furthermore, our energy footprint is smaller, making up 11% of the total cost. Comparing this to the most and least expensive models, gives an overhead of 7% and 25%, respectively.

Machine translation. Feature extraction costs are much smaller in this domain, hence the overheads of our premodel are negligible: <6ms overall, which accounts for 0.5% of the end-to-end cost when translating a sentence. Similarly, the energy cost of our premodel accounts for 0.48% of the overall energy cost. The memory footprint of our premodel is also insignificant.

7.5 Soundness Analysis

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. Theoretical proof of soundness guarantee of machine learning models is an outstanding challenge and is out of the scope of the paper [4]. Nonetheless, there are two possible ways to empirically estimate the prediction confidence: (1) using the distance on the feature space as a soundness measurement, or (2) using statistical assessments. We described both methods as follows.

Distance measurement. Figure 17 shows how the accuracy of image classification (under the top-1 score) changes as the permissible distance for choosing the nearest training images changes. Recall that each training image is associated with an optimal model for that image and by choosing the nearest training images to the input, we can then use a voting scheme to determine which of the associate DNNs to use for the input image. Here, the distance is calculated by computing the Euclidean distance between the input testing image and a training image on the feature space. The results are averaged across our testing images using cross-validation. When the permissible distance increases from 0 to 2, we see an increase in the inference accuracy. This is because using a short distance reduces the chance of finding a testing image that is close enough. However, we observe that when the permissible distance is greater than 2, the inference accuracy drops as the distance increases. This is because when the permissible distance goes beyond this point, we are more likely to choose a testing image (and the associated optimal model) that is not similar enough to the input. This example shows that the permissible distance can be empirically determined and used as a proxy for the accuracy confidence.

Statistical assessments. Another method for soundness guarantee is to combine probabilistic and statistical assessments. This can be done by using a Conformal Predictor (CP) [62] to determine to what degree a *new, unseen* input conforms to previously seen training samples. The CP is a statistical assessment method for quantifying how much we could trust a model's prediction. This is achieved by learning a nonconformity function from the model's training data. This function estimates the "strangeness" from input features, x, to a prediction output, y, by looking at the input and the probability distribution of the model prediction. Specifically, we learn a nonconformity function, f, from our premodel training dataset, which produces a non-conformity score for the premodel's input x_i and output y_i:

$$f(x_i, y_i) = 1 - \hat{P}_h(y_i|x_i)$$

Here, $\hat{P}_h$ is the statistical distribution of the premodel's probabilistic output, calculated as:

$$p_{x_i}^{y_i} = \frac{|\{z_j \in Z : a_j > a_i^{y_i}\}|}{q+1} + \theta \frac{|\{z_j \in Z : a_j = a_i^{y_i}\}| + 1}{q+1}, \theta \in [0, 1]$$

where Z is part of the training dataset chosen by the CP, q is the length of Z, a_i is the calibration score learned from training data, $a_i^{y_i}$ is the statistical score for premodel prediction y_i, and θ is a calibration factor learned by the CP.

The learned function f produces a non-conformity score between 0 and 1 for every class for each given input. The closer the score to 0, the more likely the input is to conform to the premodel's output, *i.e.* it is similar to training samples of that class. By choosing a threshold, we can predict whether our premodel gives an incorrect DNN for a given input. By implementing an SVM based conformal predictor for image classification, and using a threshold value of 0.5, we can correctly predict when our premodel will choose an incorrect DNN 87.4% of the time, with a false positive rate of 5.5%. This experiment shows that we can use the CP to estimate if the premodel's output can be trusted to provides a certain degree of soundness guarantee.

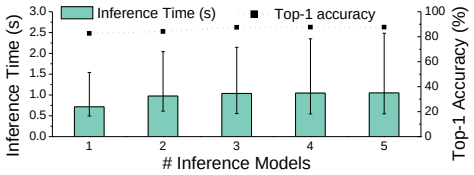


Fig. 18. Overhead and achieved performance when using a different number of DNN models. The range of inference time across testing images is shown using min-max bars.

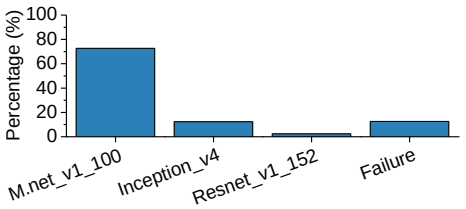
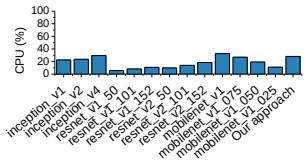
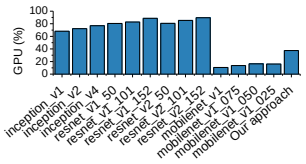


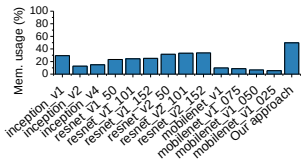
Fig. 19. The utilization of each DNN included in our `premodel`.



(a) CPU Utilization



(b) GPU Utilization



(c) Memory Utilization

Fig. 20. The average CPU, GPU, and Memory utilization per model. Compared against our approach.

7.6 Further In-Depth Analysis

This section contains an in-depth analysis using image classification as a case study. The results are similar when we apply the same analysis to the machine translation case study.

7.6.1 Changing the premodel Size. In Section 3.3 we describe the method we use to chose which DNN models to include. Using the *Accuracy* method, and temporarily ignoring the model selection threshold θ in Algorithm 1, we constructed Figure 18, 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 (≈87.5%) by using three KNN 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 θ. Additionally, Figure 19 shows the utilization percentage of each model by our approach. Our approach can also choose to not select any model for an image if it deems none of the available models as suitable for it. We use *Failure* to represent this in the Figure. Overall, 87.5 % of the time a model is selected, leaving 12.5 % of the time *Failure* is selected.

In Section 3.3 we describe the method we use to chose which DNN models to include. Using the *Accuracy* method, and temporarily ignoring the model selection threshold θ in Algorithm 1, we constructed Figure 18, 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 (≈87.5%) by using three KNN 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 θ. Additionally, Figure 19 shows the utilization percentage of each model by our approach. Our approach can also choose to not select any model for an image if it deems none of the available models as suitable for it. We use *Failure* to represent this in the Figure. Overall, 87.5 % of the time a model is selected, leaving 12.5 % of the time *Failure* is selected.

7.6.2 Resource Utilization. Figure 20 shows the average CPU, GPU and memory utilization of a selection of image classification DNNs. We recorded the utilization of each resource during inference on every image in the ImageNet ILSVRC 2012 *validation dataset*, and report the average.

Table 9. The change in model size when using compression on *Resnet_v2_152*

Model	Size (MB)
Without Compression	691
Deep Compression	317.12
Quantization	473.42
Both Compression Methods	226.22

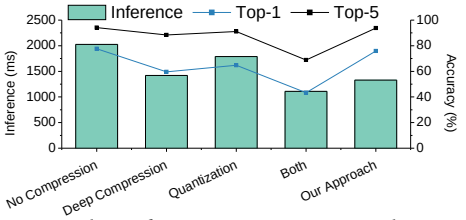


Fig. 21. The inference time, top-1, and top-5 performance using compression on a single DNN.

CPU. Figure 20a shows the CPU utilization. All DNNs primarily run on the GPU, therefore we see a low CPU utilization overall; no DNN has a utilization higher than 30%. Our approach is one of the most expensive, using 28.11% of the CPU, it is only cheaper than `MobileNet_v1` and `Inception_v4`, which use 32.63% and 29.42%, respectively. In this category, our approach is expensive as we include the two most expensive models.

GPU. GPU utilization is shown in Figure 20b. As expected, this is much higher than CPU utilization, with the majority of DNNs using between 70-90% of the GPU. In contrast, our approach has a much lower utilization of 37.46%; 52.18% lower than the most expensive model, `ResNet_v2_152`. We achieve this by making use of `MobileNet_v1` whenever possible, which has a utilization of 10.57%.

Memory. Figure 20c compares the memory utilization. Our approach keeps the selected DNNs in memory, therefore it is the most expensive in this category. However, our approach only requires 16% more memory than the most expensive model, a small cost to pay for reduced CPU and GPU load, and a faster inference time with higher accuracy.

7.6.3 Compression. So far we have shown the ability of our approach to utilize multiple DNNs, however, this is not always possible. In some cases only a single trained DNN is available, this could be caused by a number of reasons, *e.g.* limited training time. This section shows how our approach can still be utilized in this case, by making use of compression. We use two different compression algorithms: Deep Compression [24] and Quantization [33]. By first applying Deep Compression followed by Quantization, we effectively have a third compression "algorithm". Compression is designed to make a DNN *lighter* – it has a faster inference time and a smaller size (See Table 9) – however, as a consequence the model accuracy also degrades.

We chose *Resnet_v2_152* as a starting model. It is the most complex model we consider with the highest accuracy, unfortunately, as a consequence it also has the longest runtime at 2026ms. By applying each of our three compression algorithms, we generate a total of 4 *different* models. Using the 4 distinct DNNs, we apply our method to create a new `premodel`.

Figure 21 shows the performance of each compressed model and our approach. There is a clear trend, applying compression reduces accuracy while reducing inference time. Applying both compression methods in practice would result in an unacceptable accuracy drop, reducing top-1 accuracy by 34.32%. However, it makes sense in this scenario as our approach is able to make use of a model compressed by both methods when it can meet the accuracy constraint. Our approach is able to achieve a minor drop in accuracy (1.76% for top-1, and 0.31% for top-5), while reducing inference time by 1.52x. Effectively, we are able to utilize the positive of compression (reduced runtime) while keeping the accuracy of the original model.

8 DISCUSSION

Feature extraction. The majority of our image classification 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 hotshots in our code which would benefit from parallelism.

Premodel training. There is room for improvement for our machine translation `premodel`. We were unable to reach the full potential shown by the `Oracle`. To aid the `premodel` in reaching its full potential would require improving its accuracy. We believe we require more training data. There was only 5K sentences available for machine translation, in comparison to 50k images for image classification.

Computation Offloading. This work focuses on accelerating inference on the current device. Future work could involve an environment with the opportunity to offload some of the computation to either cloud servers, or other devices at the edge [15]. Accomplishing this would require a method to measure and predict network latency, allowing an educated decision to be made at runtime. ML techniques are shown to be effective in learning a cost function for profitability analysis [22]. This can be integrated with our current learning framework.

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 ML techniques to be successful at selecting the optimal computing device [67]. This can be integrated into our existing learning framework.

Model size. Our approach uses multiple pre-trained `DNN` models for inference, in comparison, the default method is to simply use a single model. Therefore, our approach requires more storage space. A solution for this would involve using model compression techniques to generate multiple compressed models from a single, highly accurate model. We have shown that our approach is effective at choosing between compressed models. The result of this is numerous models sharing many weights in common, which allows us to amortize the cost of using multiple models.

9 RELATED WORK

Methods have been proposed to reduce the computational demands of a deep model by trading prediction accuracy for runtime, compressing a pre-trained network [9, 25, 53], training small networks directly [19, 54], or a combination of both [30]. Using these approaches, a user now needs to decide when to use a specific model. Making such a crucial decision is a non-trivial task as the application context (*e.g.* the model input) is often unpredictable and constantly evolving. Our work alleviates this user burden by automatically selecting an appropriate model to use.

Neurosurgeon [35] 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 work presented by Rodríguez et al. [57] 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 `CNN`s on embedded devices. They aim to accelerate inference time by exploiting parameter tuning [40], computational kernel optimization [5, 26], task parallelism [47, 52], and trading precision for time [31] etc. Since a single model is unlikely to meet all the constraints of accuracy, inference time and energy consumption across inputs [23], 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 [69], 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 [50]. 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.

Machine learning has been employed for various optimization tasks, including code optimization [7, 8, 10, 11, 22, 48, 49, 67, 70, 74–79, 81], task scheduling [12, 16, 20, 21, 45, 55, 56], cloud deployment [59, 60], network management [72], 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 [17], wireless sensing [80], and optimize application memory usage [46]. This work is the first attempt in applying this technique to optimize deep inference on embedded devices.

Many significantly notable improvements have been made for machine translation over the last few years, including Google Neural Machine Translation [18], and the introduction of the Attention architecture [73]. A common method to improve machine translation accuracy is ensembling [61, 65], where multiple models are used during one translation. Our approach is able to see improvements in accuracy without the added cost of ensembling; we only run one translator model for each translation task. In recent years `CNNs` have become the norm for sentence classification. [37] shows that even simple `CNNs` can be used classify sentences with high accuracy, however running a `CNN` on embedded systems is expensive. Joulin et al. [34] explore a simple, fast text classifier. Unfortunately, this classifier leads to poor performance on our data.

10 CONCLUSION

We have presented a novel approach for efficient deep learning inference for embedded systems. Our approach leverages multiple `DNNs` through the use of a `premodel` that dynamically selects the optimal model to use, depending on the model input and evaluation criterion. We developed an automatic approach for `premodel` generation as well as feature selection and tuning. We apply our approach to two deep learning domains: image classification and machine translation, which involve convolutional and recurrent neural network architectures. Experiment results show that our approach deliver portable good performance across application domains and neural network architectures. For image classification, 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. For machine translation, our approach is able to reduce inference time by 1.34x than the single most capable model, without significantly effecting accuracy. With more training data we could achieve the same reduction in accuracy while increasing F1 measure by 20.51%.

ACKNOWLEDGEMENT

This work was partly supported by the UK EPSRC under grants EP/M015734/1 (Dionasys) and EP/M01567X/1 (SANDeRs). For any correspondence, please contact Zheng Wang (E-mail: z.wang5@leeds.ac.uk).

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*.
[3] Dzmitry Bahdanau et al. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[4] Jiawang Bai et al. 2019. Rectified Decision Trees: Towards Interpretability, Compression and Empirical Soundness. (2019). arXiv:1903.05965
[5] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *SenSys*.
[6] Alfredo Canziani et al. 2016. An Analysis of Deep Neural Network Models for Practical Applications. *CoRR* (2016).
[7] Donglin Chen et al. 2019. Characterizing Scalability of Sparse Matrix-Vector Multiplications on Phytium FT-2000+. *International Journal of Parallel Programming* (2019).
[8] Shizhao Chen et al. 2018. Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures. In *HPCC '18*.

[9] Wenlin Chen et al. 2015. Compressing Neural Networks with the Hashing Trick. In *ICML*.
[10] Chris Cummins et al. 2017. End-to-end Deep Learning of Optimization Heuristics. In *PACT*.
[11] Chris Cummins et al. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*.
[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*.
[13] Jeff Donahue et al. 2014. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In *ICML*.
[14] Yehia Elkhatib. 2015. Building Cloud Applications for Challenged Networks. In *Embracing Global Computing in Emerging Economies*. Communications in Computer and Information Science, Vol. 514.
[15] Yehia Elkhatib et al. 2017. On Using Micro-Clouds to Deliver the Fog. *Internet Computing* 21, 2 (March 2017), 8–15.
[16] Murali Krishna Emani et al. 2013. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO '13*.
[17] Murali Krishna Emani and Michael O'Boyle. 2015. Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. In *PLDI*.
[18] Yonghui Wu et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016).
[19] 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).
[20] Dominik Grewe et al. 2011. A workload-aware mapping approach for data-parallel programs. In *HiPEAC '11*.
[21] Dominik Grewe et al. 2013. OpenCL task partitioning in the presence of GPU contention. In *LCPC '13*.
[22] Dominik Grewe et al. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *CGO*.
[23] Tian Guo. 2017. Towards Efficient Deep Inference for Mobile Applications. *CoRR* abs/1707.04610 (2017).
[24] Song Han et al. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* (2015).
[25] Song Han et al. 2015. Learning both weights and connections for efficient neural network. In *NIPS*.
[26] Song Han et al. 2016. EIE: efficient inference engine on compressed deep neural network. In *ISCA*.
[27] M Hassaballah et al. 2016. Image features detection, description and matching. In *Image Feature Detectors and Descriptors*.
[28] Kaiming He et al. 2016. Deep residual learning for image recognition. In *CVPR*.
[29] Kaiming He et al. 2016. Identity mappings in deep residual networks. In *ECCV*.
[30] Andrew G. Howard et al. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[31] Loc N. Huynh et al. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *MobiSys*.
[32] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*.
[33] Benoit Jacob et al. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*.
[34] Armand Joulin et al. 2017. Bag of Tricks for Efficient Text Classification. In *EACL*.
[35] Yiping Kang et al. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS*.
[36] Yuval Marom Anthony Khoo and David Albrecht. 2006. Experiments with sentence classification. In *The Australasian Language Technology Workshop*.
[37] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).
[38] Aaron Klein et al. 2016. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079* (2016).
[39] Nicholas D Lane and Pete Warden. 2018. The deep (learning) transformation of mobile and embedded computing. *Computer* 51, 5 (2018), 12–16.
[40] Seyyed Salar Latifi Oskouei et al. 2016. Cnndroid: GPU-accelerated execution of trained deep convolutional neural networks on android. In *Multimedia Conference*.
[41] Tsung-Yi Lin et al. 2014. Microsoft coco: Common objects in context. In *ECCV*.
[42] Marco Lui. 2012. Feature stacking for sentence classification in evidence-based medicine. In *The Australasian Language Technology Association Workshop*.
[43] Minh-Thang Luong et al. 2017. Neural Machine Translation (seq2seq) Tutorial. *https://github.com/tensorflow/nmt* (2017).
[44] Walid Magdy et al. 2017. Fake it till you make it: Fishing for Catfishes. In *ASONAM*.
[45] Vicent Sanz Marco et al. 2017. Improving spark application throughput via memory aware task co-location: a mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*.

[46] Vicent Sanz Marco et al. 2017. Improving Spark Application Throughput via Memory Aware Task Co-location: A Mixture of Experts Approach. In *Middleware*.

[47] 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).

[48] William F Ogilvie et al. 2014. Fast automatic heuristic construction using active learning. In *LCPC '14*.

[49] William F Ogilvie et al. 2017. Minimizing the cost of iterative compilation with active learning. In *CGO '17*.

[50] Seyed Ali Ossia et al. 2017. A Hybrid Deep Learning Architecture for Privacy-Preserving Mobile Analytics. *CoRR* abs/1703.02952 (2017).

[51] Omkar M Parkhi et al. 2015. Deep Face Recognition. In *BMVC*.

[52] Sundari K. Rallapalli et al. 2016. *Are Very Deep Neural Networks Feasible on Mobile Devices?* Technical Report. University of Southern California.

[53] Mohammad Rastegari et al. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016).

[54] Sujith Ravi. 2015. ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections. *arXiv:1708.00630* (2015).

[55] Jie Ren et al. 2017. Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *INFOCOM*.

[56] Jie Ren et al. 2018. Proteus: Network-aware Web Browsing on Heterogeneous Mobile Systems. In *CoNEXT '18*.

[57] Sandra Servia Rodríguez et al. 2017. Personal Model Training under Privacy Constraints. *CoRR* abs/1703.00380 (2017).

[58] Olga Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. In *IJCV*.

[59] Faiza Samreen et al. 2016. Daleel: Simplifying Cloud Instance Selection Using Machine Learning. In *NOMS*.

[60] Faiza Samreen et al. 2019. Transferable Knowledge for Low-cost Decision Making in Cloud Environments. (2019). arXiv:1905.02448

[61] Danielle Saunders et al. 2018. Multi-representation ensembles and delayed SGD updates improve syntax-based NMT. *arXiv* (2018).

[62] Glenn Shafer and Vladimir Vovk. 2008. A tutorial on conformal prediction. *Journal of Machine Learning Research* 9, Mar (2008), 371–421.

[63] Nathan Silberman and Sergio Guadarrama. 2013. TensorFlow-slim image classification library. https://github.com/tensorflow/models/tree/master/research/slim. (2013).

[64] Mingcong Song et al. 2017. Towards pervasive and user satisfactory CNN across GPU microarchitectures. In *HPCA*.

[65] Felix andothers Stahlberg. 2018. The University of Cambridge's Machine Translation Systems for WMT18. *arXiv* (2018).

[66] Yi Sun et al. 2014. Deep learning face representation by joint identification-verification. In *NIPS*.

[67] Ben Taylor et al. 2017. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. In *LCTES*.

[68] Ben Taylor et al. 2018. Adaptive deep learning model selection on embedded systems. In *LCTES*. ACM, 31–43.

[69] Surat Teerapittayanon et al. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *ICDCS*.

[70] 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*.

[71] EMNLP 2015 TENTH WORKSHOP ON STATISTICAL MACHINE TRANSLATION. [n. d.]. Shared Task: Machine Translation. ([n. d.]). https://www.statmt.org/wmt15/translation-task.html

[72] Muhammad Usama et al. 2017. Unsupervised Machine Learning for Networking: Techniques, Applications and Research Challenges. *CoRR* abs/1709.06599 (2017).

[73] Ashish Vaswani et al. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

[74] Zheng Wang et al. 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM TACO* (2014).

[75] Zheng Wang et al. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM TACO* (2014).

[76] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimisation. *Proc. IEEE* (2018).

[77] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *PPoPP '09*.

[78] Zheng Wang and Michael FP O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*.

[79] Zheng Wang and Michael FP O'boyle. 2013. Using machine learning to partition streaming programs. *ACM TACO* (2013).

[80] Jie Zhang et al. 2018. CrossSense: Towards Cross-Site and Large-Scale WiFi Sensing. In *MobiCom '18*.

[81] Peng Zhang, , et al. 2018. Auto-tuning Streamed Applications on Intel Xeon Phi. In *IPDPS '18*.