Mistify: Automating DNN Model Porting for On-Device Inference at the Edge

Peizhen Guo Bo Hu Wenjun Hu
Yale University

Abstract

AI applications powered by deep learning inference are increasingly run natively on edge devices to provide better interactive user experience. This often necessitates fitting a model originally designed and trained in the cloud to edge devices with a range of hardware capabilities, which so far has relied on time-consuming manual effort.

In this paper, we quantify the challenges of manually generating a large number of compressed models and then build a system framework, *Mistify*, to automatically port a cloud-based model to a suite of models for edge devices targeting various points in the design space. *Mistify* adds an intermediate "layer" that decouples the model design and deployment phases. By exposing configuration APIs to obviate the need for code changes deeply embedded into the original model, *Mistify* hides run-time issues from model designers and hides the model internals from model users, hence reducing the expertise needed in either. For better scalability, *Mistify* consolidates multiple model tailoring requests to minimize repeated computation. Further, *Mistify* leverages locally available edge data in a privacy-aware manner, and performs run-time model adaptation to provide scalable edge support and accurate inference results. Extensive evaluation shows that *Mistify* reduces the DNN porting time needed by over 10× to cater to a wide spectrum of edge deployment scenarios, incurring orders of magnitude less manual effort.

1 Introduction

AI-driven intelligent edge has already become a reality [9], where millions of mobile and IoT devices or edge servers analyze real-time data and transform those into actionable insights on user-facing devices. For example, real-time video analytics (e.g., traffic monitoring [43], security surveillance [5], and smart retail [3]), natural language understanding (e.g., virtual assistance, smart email composition [63]), visual assistance [48], and industrial automation (e.g., defect detection, assembly line management [1, 5]) are already everyday examples. It is projected that, by 2022, over 60% of the data locally generated by IoT, sensor, and mobile devices will drive real-time intelligent decisions; 80% of the IoT and mobile devices shipped will have on-device AI capabilities [6, 16].

Many AI functionalities today are powered by deep learning (DL), with a significant computation footprint. While edge devices used to primarily offload related computation to the cloud, increasingly inference workloads are run natively on the edge devices to provide better interactive user experience (e.g., ∼10 ms real-time response), data privacy, and reliability [60]. This often necessitates *porting* (i.e., *tailoring* and deploying) a deep neural network (DNN) model originally designed and trained on the cloud to edge settings.

Model porting is a non-trivial process even for a single target. From an *algorithmic* perspective, the core techniques involved are called *model tailoring* in the machine learning literature. There are two steps, adapting the architecture of a pre-trained model to fit a new resource specification, followed by fine-tuning the new model parameters. Although there have been numerous model tailoring *algorithms* [22, 25, 31, 82], the complete porting process additionally requires "executing" the algorithms by correctly annotating a source model, and then retraining the annotated model with the right data. By various estimates, there will be over 50 billions IoT devices [30] with very diverse hardware profiles. This creates a massive design space for optimizing the resource usage and performance of a new model.

Unfortunately, the current practice of porting relies on manual effort, which simply cannot scale with the sheer size of the design space. There are two issues: laborious manual annotations and the computational complexity. Even if porting is a one-time need, it takes time to meticulously annotate the original model to embed the correct model tailoring objectives. For instance, constructing the model tailoring logic for ResNet50 [35] requires around 30 lines of source code edits scattered around several files. Further, model adaptation incurs significant computational complexity. Existing algorithms can handle generating one model, but can not scale well to large batches of model generation. If many model variants are needed, either for different device hardware specifications or for different runtime conditions, manual tailoring incurs significant repeated efforts. The effort needed to tailor model variants could match that for training an original model. Therefore, app developers currently perform little platform-specific customization to the intractable target space [75], even though lack of customization results in suboptimal performance. (Section 2)

Fundamentally, the problem is the implicit coupling between model design and deployment currently. Model designers need to both improve the inference accuracy and minimize the memory and computation footprint for deployment. Model users need to both compress the model without degrading accuracy significantly and accelerate the inference. Both

stages require significant expertise spanning deep learning algorithms and system runtime management.

In this paper, therefore, we build *Mistify*, a system *framework* to automate and scale the porting process from a pre-trained model to a suite of compact models tailored to diverse edge resource specifications (Section 3). *Mistify* does not propose any new model tailoring algorithm. Instead, it wraps over existing model tailoring algorithms and provides additional system services for scalability. This is analogous to a scheduling framework implementing common scheduling algorithms so that application developers can outsource scheduling considerations. With *Mistify*, model users (i.e., mobile app developers, often non-machine-learning experts) can outsource model adaptation instead of understanding the specifics of the model to adapt. In this way *Mistify* takes on the role of a provider of models for on-device inference execution. Meanwhile, model designers (i.e., machine learning experts) can use simple resource abstractions to evaluate the model instead of undergoing detailed resource profiling.

From a *system* perspective, we propose new abstractions to separate the model semantics from the execution characteristics and several techniques (*Collective adaptation*, *Privacy-aware knowledge distillation*, and *Downtime-free run-time model generation and switching*), all incorporated in an end-to-end framework. *Mistify* minimizes the need for "compile-time" code changes when generating a model statically. Instead of requiring the user to annotate the original model, *Mistify generates* model adaptation logic from the configuration file to correctly and scalably tailor to the resource budgets and performance requirements of each device while minimizing duplicate iterations (Section 4). Further, *Mistify* leverages implicitly correlated edge data in a privacy-aware manner to balance training data privacy and model accuracy (Section 5). During the run time of the inference task, *Mistify* employs a feedback mechanism to generate new models as needed to adapt to fluctuating application demands and resource availability (Section 6).

Note that we make a distinction between the end-to-end process (*porting*) and individual model compression techniques (*neural architecture search*, *layer pruning*, etc.). The latter do not always produce a readily usable compressed model. The (adapted) model still needs (re)training and that process incurs several practical difficulties. In contrast, *Mistify* automates the entire end-to-end process. Making it scalable and adaptive while keeping training data local are non-trivial efforts, and extend significantly beyond simply implementing known algorithms for each component in one system.

Mistify is implemented following a client-server model, built on TensorFlow [13]. We build example wrappers to adopt state-of-the-art model adaptation algorithms like MorphNet [31] and ChamNet [25], and evaluate *Mistify* using representative vision and natural language processing (NLP) models trained with widely used standard datasets. Extensive evaluation shows that *Mistify* reduces the DNN porting time needed by over 10× and incurs orders of magnitude less manual effort, all with little or up to 1% accuracy loss when compared to manually running the adaptation algorithms. This loss margin is well within the typical accuracy loss budget for on-device inference [64].

Mistify is far more than a tool for convenience. It serves as an intermediate layer that decouples the model design and deployment stages. Model designers can focus on model performance and advanced architecture design, without worrying about deployment difficulties, whereas edge users can focus on execution-centric issues such as optimizing the executable binaries, computation kernels, and job scheduling, without worrying about the inner workings of the model.

To summarize, this paper makes three contributions: First, we quantify the scalability challenges of porting pre-trained DNN models to edge settings to motivate framework support. Second, we design and implement *Mistify* as a framework for automated porting at scale. *Mistify* achieves scalability with collective adaptation and improves model quality with privacy aware knowledge distillation and run-time model adaptation. Third, *Mistify* provides a clean interface to separate DNN model design and deployment. This could lower the bar for wider usage of on-device deep learning at the edge.

2 Background and motivation

The lifecycle of a DNN model spans design and deployment, and the need for automating model porting arises from the complexity of the process. We discuss these in detail before outlining the challenges and solutions.

2.1 Current DNN lifecycle

The lifecycle of a DNN encompasses at least three stages: model design, publishing, and deployment. Publishing mainly requires adding a well-trained model to public repositories, while design and deployment are more involved.

DNN model design. Today's models are designed for either optimal inference quality or minimal resource footprint.

The former is typically assumed for workloads run on the cloud. Given increasing computation power, cloud-centric models employ advanced neural network topologies, millions of parameters and floating-point operations (FLOPs) to achieve the *highest accuracy*. For example, BERT [28] and ResNeXt [51] have 340 and 829 million parameters respectively, hence extremely computation intensive.

The latter goal is geared towards resource-constrained edge devices, including IoT nodes, smartphones and tablets. The desirable models (e.g., MobileNet [38] and SqueezeNet [40]) are exceedingly compact, requiring only a few MBs for storage and affordable computing budget, ready to run across diverse hardware. However, these DNNs sacrifice accuracy in exchange for super lightweight execution, aiming at *maximal deployment coverage*.

DNN deployment at the edge. Many DL inference engines

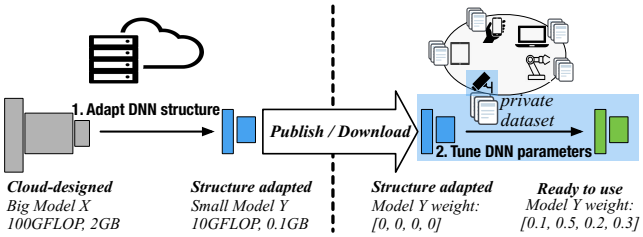


Figure 1: Steps to port a DNN model to an edge setting.

have been developed to serve DNN workloads on edge devices. They focus on deployment optimizations such as cross-platform compatibility, trimming executable size, and low-complexity operator kernels [2, 24]. Once DNN models are loaded (e.g., from model repositories or custom URLs), these engines can execute the inference tasks efficiently.

Transition from design to deployment. When a pre-trained model is ill-suited to a desirable deployment setting, it needs to be *tailored* to the new resource budget and performance goals. This requires *adapting* the model architecture (e.g., by trimming network connections, skipping layers, quantizing parameters) and then *fine-tuning* (i.e., retraining) the parameters with local datasets (Figure 1). However, the end-to-end model porting process is complex. The source model needs to be correctly annotated to have its architecture adapted to a desired setting. Fine-tuning also requires careful usage of the training data to balance training quality (i.e., effective specialization without overfitting) and data privacy.

2.2 The complexity of porting DNN models

As more edge devices adopt on-device inference, porting cloud-based models to edge settings becomes increasingly complex, facing several challenges: (i) the range of model adaptation targets is huge as a result of the diversity in the hardware specification; (ii) the porting process involves several stages, each requiring coordination between multiple parties; (iii) run-time dynamics and new deployment settings may necessitate frequent model re-adaptations.

Heterogeneous execution environment. Edge devices are incredibly diverse, ranging from embedded sensors, IoT devices, mobile phones/tablets, to edge servers, covering a full spectrum of hardware capability [75]. Table 1 lists the specifications of some GPU and ASIC accelerators and processors, from high-end to low-end, widely employed at the edge for DNN-based workloads. For the same DNN inference workload, the completion times for low-end (e.g., Jetson nano) and high-end (e.g., 2080) devices can differ by orders of magnitude (e.g., 229 ms vs 9.8 ms to run inference with ResNet).

Meanwhile, the DNN inference times for the same task can differ by up to 8×, and the quality (e.g., in *accuracy*, *F1 score*) varies by as much as 25% [18, 66]. It is therefore essential to match the desirable performance with the hardware capability.

These numbers outline a massive design space to explore different tradeoff points between inference accuracy and latency, where a sub-optimal choice could incur up to 10% accuracy loss (e.g., when running EfficientNet-BO unnecessarily on the latest iPhone model) or miss the latency requirement for real-time processing by over 100 ms (e.g., running ResNet on a low-end smartphone) [76].

Table 1: Popular DL hardware specifications.

GPU	Peak perf	Memory	Bandwidth
V100	112 TFLOP	32 GB	900 GB/sec
2080	11.7 TFLOP	11 GB	480 GB/sec
Edge GPU	**Peak perf**	**Memory**	**Bandwidth**
Jetson TX2	1.5 TFLOP	4 GB	58 GB/sec
Jetson nano	0.47 TFLOP	4 GB	25 GB/sec
ASIC	**Peak perf**	**Memory**	**Bandwidth**
Edge TPU [4]	4 TFLOP	-	-
Raspberry pi	6 GFLOP	2 GB	8.5 GB/sec

Clearly, one size does not fit all, but nor would a few sizes only. Instead, it is desirable to tailor to each target at a fine granularity. For instance, EfficientNet-B4 (a popular model occupying a sweet spot of computation complexity and prediction accuracy) is suitable for Samsung S9, achieving 83% accuracy and 50 fps real-time response rate. However, using the same DNN on its immediate predecessor (S8) and successor (S10) would reduce the response rate by 14 fps for S8 and the accuracy by nearly 1% for S10. These are significant to the model designers where even 0.1% accuracy improvement merits tremendous effort (both intellectually and computationally) into model design and training. Given the ever increasing size of this adaptation space, it is impractical to either cover all plausible operation points with a few DNN models, or manually exhaust the entire space to customize the adaptation tradeoff for each possible individual edge setting.

Multi-stage multi-party efforts. Tailoring a DNN model involves first adapting to the right model architecture, and then fine-tuning the model parameters (Figure 1).

The first stage is resource heavy and therefore takes place where the original models are trained (i.e., in the *cloud*).The second stage increasingly takes place at the *edge* given the push for on-device inference and private learning. Edge devices collect and maintain specialized data relevant to the local context for model training [19, 42] and local data are typically privacy sensitive [59]. However, smaller networks with less abundant datasets are well known to be much harder to train, as it is easy to overfit the model to the training data such that the model may not generalize well to unseen test data [37, 79]. Thus, it is also preferable for the edge to take advantage of relevant datasets available elsewhere (e.g., in the cloud or on other devices) to enhance the training dataset and improve training quality.

To sum up, both stages of model tailoring require coordination between the cloud and the edge, and resolving the conflict between data privacy and fine-tuning quality.

Fluctuating run-time characteristics. The run-time characteristics of deep learning inference tasks are highly dynamic, shown in two aspects. First, the performance require-

ments, e.g., accuracy and response time, of an inference task change frequently. For instance, the accuracy requirements of a vision-based security surveillance workload differ between crucial and trivial moments, while the latency requirements fluctuate across peak and off-peak hours (e.g., daytime and night) [41,42]. Further, the commonly used metric, FLOPS, is sometimes an inaccurate proxy to statically estimate run-time latency [72]. Second, the resource availability (e.g., memory space, CPU cycles, accelerator quotas), varies on the edge device due to other workloads competing for the same resource. For instance, when an edge device launches or completes a workload, or adjusts the resource allocation of the containers that serve the inference tasks, the perceived resource availability to the active workloads changes [23, 74].

Frequent changes in the performance requirements and resource availability necessitate a mechanism to better serve combinations of the individual operation points, including a suite of models to switch to dynamically, and asynchronously tailoring new ones as the demand warrants.

2.3 The need to automate DNN porting

Current practice. To tailor DNNs towards heterogeneous deployment settings, currently either the model designers should generate different DNNs to cater to each possible resource budget and performance goal, or the model users should prepare the custom datasets, select and apply the algorithms to tailor already published DNNs towards their custom settings.

The latest adaptation algorithms, such as AutoML [82], EfficientNet [73], and others [14,22,25,31], all address target-specific adaptation *case by case* as an additional step in the *design* phase. While the techniques differ (e.g., gradient-based, evolutionary, and recurrent neural network based), they revise the model architecture to be closer to the required resource and performance targets with successive training iterations.

Problems with current porting practice. The overarching problems of the existing practice are they do not scale from a system perspective (e.g., hundreds of GPU hours for a single setting [71, 82]) and largely rely on manual effort (e.g., thousands of lines of code spread across source files [10]). Such a manual tailoring process is not easily turned to a configuration style that is agnostic to the number of cases because distinct structure adapting terms have to be added to different DNN models/layers and at specific positions, which makes it difficult and error-prone. Furthermore, it is infeasible for the model designers to prepare for all possible deployment settings, or for the model users to be well versed in machine learning literature to run the right algorithm.

The need for an automated framework. The current model porting process implicitly couples DNN design and deployment, even though they are conceptually separate stages. This coupling introduces unnecessary complexity to both model designers and model users. This motivates adding a separate *model porting* stage to the model lifecycle, i.e., an intermediary to decouple design from deployment and automatically port pre-trained DNNs towards heterogeneous edge settings.

Mistify is therefore built as an intermediate framework to encapsulate diverse adaptation algorithms and address the end-to-end porting challenges outlined above, analogous to scheduler *frameworks* for distributed systems implementing scheduling *algorithms* and providing services.

2.4 System requirements

To address the challenges above, an automated model porting framework should meet the following requirements.

Avoiding deeply embedded and unscalable manual code changes. Since the existing model adaptation step is often coupled with model design itself, a side effect is that relevant code changes are embedded deep into the model design code. Therefore, the system challenge is to simplify the code modifications needed to specify the adaptation logic.

Mistify addresses this challenge in two steps (Section 4). First, we expose the right high-level abstractions of adaptation choices to users. This elevates per-model code edits (embedded in the particular script specifying the model) to framework level configuration parameter changes. Second, we parse the adaptation requirements from the configuration files and merge implicitly correlated model adaptation requests to reduce duplicate iterations and improve scalability.

Cloud-edge coordination. To automate the two-stage model tailoring process with the best training outcome, the main challenge is to simultaneously ensure that private data stay local but parameter tuning can benefit from the data distributed across devices. We address this by adopting mutual knowledge distillation. Our system implicitly coordinates multiple devices in the same tailoring batch to maximally "share" available training data in a privacy aware fashion, without explicitly exchanging and examining the raw data (Section 5).

Fast response to run-time dynamics. Fundamentally, the system challenge is to effectively handle the mismatch between a statically trained model and the dynamic execution environments during run time. Specifically, this requires generating new models as needed and switching to them with minimal downtime. We address the challenge with a feedback mechanism between the model deployment points (e.g., edge devices) and the model tailoring point (e.g., a central server or cloudlet) to perform real-time DNN re-adaptation (Section 6).

3 *Mistify* demystified

The overarching goal for *Mistify* is two-fold: (i) *Mistify* should separate the model design and deployment stages with a clean interface; and (ii) *Mistify* should bridge the two stages with a framework that automatically explores the design space at scale and generates models best suited to user-specified tradeoff points, hiding such complexity from both sides.

Therefore, *Mistify* is designed as an intermediate framework between DNN model design toolkits and deployment

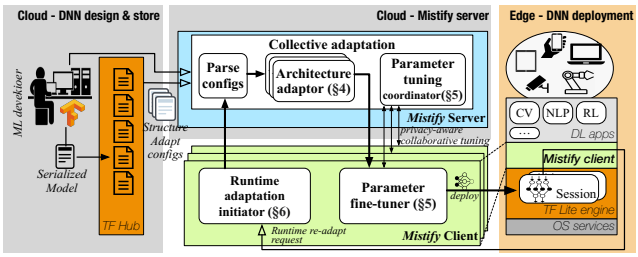


Figure 2: *Mistify* system architecture.

engines, as shown in Figure 2. The arrows across different shaded blocks show how *Mistify* interacts with model designers and users. *Mistify* exposes APIs to the model users and inference engines to specify their *porting configurations* (Figure 3), either in a *batch* mode during initialization or in a *streaming* mode incrementally during run time.

The primary challenge for *Mistify* is therefore how to generate a large number of adapted DNN models with minimal computation and manual intervention. In general, our approach is *collective adaptation*, i.e., parsing adaptation goals and harnessing the implicit correlation among the goals to reduce unnecessary computation (Section 4). *Mistify* parses a collection of individual adaptation goals into a dependency tree with each node corresponding to a distinct goal, so that each goal is adapted only from its immediate parent via a desirable, automatically selected adaptation algorithm. Next, the adapted models are distributed to the endpoints, where the *Mistify* client runtime will prepare the deployment of the adapted model by fine-tuning the parameters (Section 5). Finally, the models start running on edge devices, and the *Mistify* client monitors the execution environment (e.g., resource availability and desirable performance goals). The *Mistify* client will trigger on-demand model re-adaptation asynchronously when the environmental changes warrant a new model (Section 6).

Example deployment strategies. Following the common practice of on-device DL inference deployment [2,5], *Mistify* can be deployed in two ways. The *Mistify* server can be deployed in the cloud by the DNN application developers, interfacing with the model repository (e.g., TF-Hub) and exposing APIs to the public. Alternately, the server can be maintained by the model users (e.g., edge device administrators) in their private clouds to serve local devices (e.g., IoT nodes). *Mistify* clients are simply deployed on the edge devices as a module extension to the native DL engine.

***Mistify* server.** The *Mistify* server consists of two functional modules: an *architecture adaptor* and a *parameter tuning coordinator*. Once the *architecture adaptor* receives the original DNN model and the adaptation settings from the model users and/or the *Mistify* client, it generates the adapted models and sends those to the corresponding clients (Section 4). The *parameter tuning coordinator* serves as the central point to coordinate the parameter fine-tuning process across the *Mistify* clients (Section 5.2), whereas the actual tuning logic is executed on each client locally (Section 5.1).

```
[ // start of all configurations
    {
        "id": "1",
        "model": "Resnet",
        "dataset": {
            "train": "/path/to/train",
            "test": "/path/to/test"
        },
        "algorithm": {
            "name": "Morphnet",
            "config": {
                "threshold": 0.1,
                "init_reg_strength": 1e-9
            }
        },
        "adaptation_goal": {
            "latency": "30ms",
            "accuracy": 0.80,
            "FLOP": "5G",
            "num_of_params": "20M"
        }
    },
    // more configurations ...
    // ... more configurations,
    {
        "id": "9",
        "model": "Efficientnet",
        "dataset": {
            "train": "/path/to/train",
            "test": "/path/to/test"
        },
        "algorithm": {
            "name": "Chamnet",
            "config": {
                "init_population": 10,
                "crossover_rate": 0.7,
                "mutation": 0.08
            }
        },
        "adaptation_goal": {
            "latency": "30ms",
            "accuracy": 0.80,
            "FLOP": "5G",
            "num_of_params": "20M"
        }
    }
] // end of all configurations
```

Figure 3: Example porting configurations.

***Mistify* client.** The *Mistify* client consists of a *run-time adaptation initiator*, a *parameter fine-tuner* and a *run-time performance monitor*. The *run-time adaptation initiator* intercepts the native DNN model loading path of the inference engine to automatically trigger model adaptation during initialization, and then listens for run-time re-adaptation requests. The *parameter fine-tuner* takes an adapted DNN model as the starting point, optimizes its parameters jointly based on the local (private) training data and the guidance from the correlated neighboring counterparts (coordinated by the *Mistify* server). This approach aims to overcome overfitting while maintaining data privacy. The *run-time monitor* tracks the current performance as well as resource availability. Once these profiles change significantly, it will trigger an online model switching as well as an offline re-adaptation request.

4 Scalable model architecture adaptation

Instead of requiring the user to manually annotate the source models, *Mistify* provides expressive configuration interfaces to specify adaptation goals and constraints (Section 4.1) and suitable abstractions to capture common algorithmic steps that meet these constraints (Section 4.2). To further scale to a large target space, *Mistify* merges adaptation instances to avoid duplicate efforts (Section 4.3) with *collective adaptation*.

4.1 Adaptation goal specification

An adaptation goal reflects the desirable inference performance given static and dynamic device conditions. We assume these goals are immutable, and any changes in the runtime conditions simply generate new goals. A user provides two sets of input: *hardware profiles* and *performance targets*. Hardware profiles mainly include *compute power* (GFLOP/s) and *memory bandwidth* (GB/s). Performance targets cover latency and accuracy requirements. The specification can be extended to support custom resource capability and performance metrics by adding the corresponding profiling libraries and tools (Section 4.2). We leverage a JSON-like format (Figure 3) to specify multiple goals in a single configuration file, which is parsed before adaptation.

We next formulate the *cost budgets* of a given DNN structure based on the specification of the adaptation goal provided by the user. In terms of computation cost, each layer contributes $C_{in} * C_{out} * S_{kernel} * S_{out}$ multiplications and additions. C_{in} and C_{out} denote the input and output channels;

S_{kernel} and S_{out} denote the convolution kernel and output size for Conv operations; for normal Matmul operations, S_{kernel} and S_{out} both equal 1 as they are equivalent to 1×1 convolution on 1×1 inputs. For memory cost, each layer contributes $C_{in} * C_{out} * S_{kernel}$ parameters (S_{kernel} is 1 for Matmul operations similarly). Combined with the quantization strategy, the total memory consumption of a neural network layer can be calculated. For latency cost, we first calculate the previous two costs C_{comp} and C_{mem} respectively. Then, we leverage the hardware specifications (peak computation power and the memory bandwidth) to translate these costs into the latency cost as $C_{mem}/mem\_bandwidth + C_{comp}/comp\_power$.

4.2 Adaptation Executor

Common DNN adaptation workflows. State-of-the-art DNN adaptation algorithms follow a similar process. They take a source DNN model and adaptation goals, and search for variants of the base model architecture that fits each scenario. The search explores a high-dimensional vector space, where each hyperparameter of the DNN (e.g., #layers, #filters, kernel size, and quantization) corresponds to a specific dimension. The search process runs iteratively until the costs of the current model optimally match the adaptation goal. Typical search strategies include evolutionary search [25,77], gradient descent [14,31], and RNN-based search [72,73].

Adaptation executor. In light of this common process, we design an abstraction, an `Adaptation Executor`, that collects all adaptation settings as a closure, and exposes three function APIs (`Init()`, `Measure()`, and `Adjust()`). `Init()` loads the adaptation settings, the model, and the constraints, and then instantiates the executor that runs the chosen adaptation algorithm (default or user specified). `Measure()` is called after each adaptation iteration to determine the *costs* of the current model (e.g., model size or accuracy). Custom metrics and profiling mechanisms can be incorporated by implementing the `Measure()` API. `Adjust()` will then tune the control knobs of the algorithms (e.g., dimension-wise step size, threshold, or learning rate) to steer the cost refinements towards the adaptation goals in an optimal direction. These APIs abstract away the inner workings of heterogeneous adaptation algorithms in a universal approach, obviating the need to directly annotate the models to embed the adaptation logic. A new adaptation algorithm can interface with *Mistify* by implementing the above APIs, and the user can specify the preferred algorithm in the configuration.

Case study: Running MorphNet via an adaptation executor. The vanilla DNN training starts with defining an accuracy loss function ($\mathcal{L}_{output}$) based on the difference between the model outputs and the ground-truth labels. The loss is back-propagated to each layer i (with parameter θ_i) as $\mathcal{L}_i(\theta_i)$. Each layer calculates the gradient of the loss, and optimizes the parameters (θ_i) iteratively by minimizing the loss via gradient descent. Namely, $\theta_i^{new} = \theta_i^{old} - \eta \cdot \nabla_{\theta_i}\mathcal{L}_i(\theta_i)$.

MorphNet (a recent gradient based search algorithm [31]) converts the resource costs of DNNs as additional penalty terms of the loss function. This way, the DNN architecture is iteratively optimized via gradient descent along with the vanilla DNN training. For instance, the "useless" weight parameters will be suppressed to zero and trimmed during training when minimizing the overall loss, as they do not contribute to reducing the accuracy loss but increase the architectural loss. The adaptation process completes when each structure-related cost (e.g., number of FLOPs) satisfies the corresponding constraint, or when the pre-defined maximal running time is reached for the non-converged cases.

Manually adapting a model using MorphNet requires several steps: (i) selecting the penalty term for each operator appearing in the DNN (e.g., the *Gamma* regularizer for `BatchNorm`), (ii) specifying the input and output operators of the model, (iii) instantiating the penalty terms with the right arguments such as the trimming threshold and the learning rate, and adding them to the overall training loss, and (iv) adding the cost monitoring operators and the termination conditions. All these steps are needed for each adaptation target, and require modifying the source code of the DNN definition and training scripts. In contrast, *Mistify* only requires users to specify the high-level configurations (e.g., the adaptation algorithm, the trimming threshold) and adaptation goals (e.g., memory usage, number of FLOPs) in a single JSON file.

To encapsulate the MorphNet algorithm in a *Mistify* adaptation executor, we implement the APIs as follows. For `Init()`, we additionally implement the operations of deriving the positions (e.g., `Conv` layers) to add architectural loss terms, essentially by first finding the input layers, and then traversing the whole DNN graph topologically along the dependencies to insert the loss terms into the corresponding layers until the outputs. `Measure()` simply calculates the resource and performance costs of a given DNN independent of the adaptation algorithms. For `Adjust()`, we implement the logic of setting the learning rate of the loss term corresponding to each resource and/or performance constraint. The implementation of these APIs is lightweight (Section 7).

4.3 Collective adaptation

Multiple adaptation goals often share similar initial steps or training iterations. Handling each adaptation goal independently is very inefficient when deploying the same model to a range of devices. Therefore, *Mistify* provides a mechanism to "merge" adaptation goals to avoid duplicating the same steps. We parse the adaptation goals into an n-ary tree structure following certain rules. Goals along a branch are fulfilled one by one serially in a single pass. We also design a tree traversal mechanism to meet the constraints (e.g., time and space usage) of all goals simultaneously.

Adaptation goals compilation. As mentioned above, each single goal consists of several resource and performance constraints, and can be abstracted as a multi-dimensional

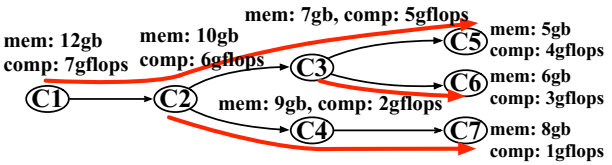


Figure 4: Example adaptation goals parsed into a tree.

vector. Combining the hardware specifications and performance constraints, we generate a partial order of all adaptation goal vectors, from the least demanding to the most. Figure 4 shows an example of 7 goals ($C1$ to $C7$), each with two constraints (memory usage `mem` and computation complexity `comp`). Goals C_i and C_j follow a strict order, $C_i < C_j$, only when $C_i.mem < C_j.mem$ and $C_i.comp < C_j.comp$.

Following the partial order between goals, we further generate a tree structure, with each node representing a goal, and each edge leading to one of its immediate more demanding goals. Hence, each branch of the tree corresponds to an independent adaptation path (marked with a red arrow in Figure 4). Along each path, every two goals are consistently ordered on all constraints. This ensures that they can be collectively adapted in one pass without conflicts. Note that the accuracy does not strictly increase over the path. When *Mistify* starts to traverse a path from one point to the next, the accuracy will first drop to a certain level, and then climb back while the training continues. Meanwhile, the resource profiles will move to the most desirable positions.

Given the tree structure, we first uniformly expand the architecture of the original DNN so that, for each constraint dimension, its actual cost value is larger than that of the root node (the least demanding goal). Then, starting from the root, we run the encapsulated adaptation algorithm to trim the DNN architecture iteratively along each adaptation path. Every time a goal is satisfied, the corresponding version of DNN is stored as a checkpoint for future use.

Note that even though the mapping between partially ordered goals to a tree structure is usually not unique, we find that there is only marginal difference in the overall adaptation time between different mappings. Hence, it is not worth optimizing the mapping given it is NP-hard.

Structure loss scheduling. When executing the adaptation along a path, an essential question is how to control the adaptation towards the optimal direction (via `Adjust()`), i.e., how to meet multiple desirable constraints simultaneously. Although this can be achieved by forking a new adaptation schedule for each change of the adaptation "direction" [69], constant forking does not scale to a large number of adaptation goals and can not achieve fine-grained continuous control.

Instead, we adjust the control knobs based on the weighted combination of the corresponding architecture losses. The overall DNN loss function is the sum of the normal loss ($\mathcal{L}$) and architecture losses corresponding to a set of constraints $\{\mathcal{G}_i\}$. For each $\mathcal{G}_i$, their control knob (e.g., learning rate for gradient-based algorithms) can be viewed as a weight parameter w_i. Hence, the overall loss $\mathcal{L}_{all} = \mathcal{L} + \sum_i w_i \cdot \mathcal{G}_i$. To adjust the adaptation "direction" towards a specific constraint f_i, we only need to increase the weight w_i of the loss $\mathcal{G}_i$.

Initially, all the weights w_i are equal and sum to 1. Suppose for the loss of constraint f_i we have the initial value $\mathcal{G}_i^{(0)}$ and the target value $\mathcal{G}_i^{(+)}$. Then, for every k training iterations (empirically set to 200), we reschedule the weights once. The n-th iteration weight $w_i^{(n)}$ is calculated as $Share_i^{(n)} / \sum_i Share_i^{(n)}$, where $Share_i^{(n)} = \frac{\mathcal{G}_i^{(+)} - \mathcal{G}_i^{(n-1)}}{\mathcal{G}_i^{(+)} - \mathcal{G}_i^{(0)}}$. In essence, we proportionally assign the next value of the weight $w_i^{(n)}$ according to how far the corresponding loss value $\mathcal{G}_i^{(n-1)}$ deviates from the target, and finally normalize these weights.

5 Privacy-aware fine-tuning at the edge

After adjusting the model architecture with respect to the resource and performance constraints, the weight parameters need to be fine-tuned before actual deployment. If all training data are collected and stored in the cloud, parameter tuning simply follows the standard training process for the adapted DNN. The challenge arises when specializing the DNNs only using the local contexts of edge devices.

Recall (Section 2.2) that DNNs are hard to train with a small dataset, usually the case for individual edge device, and can easily overfit. On the other hand, the data local to each device is often more relevant but private, making it difficult or infeasible to aggregate the data from different devices into a larger dataset for centralized training. Therefore we need to balance protecting edge data privacy and ensuring training quality (in terms of how well individual models generalize). While many works (e.g., federated learning [19] and others [20, 52, 70]) address decentralized private DNN training, they assume different endpoints train the *same DNN structure* with different local datasets. The situation is different for *Mistify*, where the models on different devices have *different architectures* to meet specific adaptation goals.

Knowledge distillation (KD). To tackle the aforementioned dilemma, we need a mechanism for DNN "knowledge" sharing between distinct peer models and without explicitly exchanging private data between devices. Fortunately, *mutual knowledge distillation* [15, 81] comes to the rescue. When training a DNN model (M_1, the *student* model) from scratch, leveraging additional help from another similar but independently trained model (M_2, the *peer teacher* model) can significantly improve the validation accuracy of M_1.

Specifically, the optimization of parameter θ_i follows: $\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, M_1(x)) + \varphi(M_2(x), M_1(x))\}$, where ∇_θ denotes taking derivatives with respect to the variable θ, ϕ and φ denote the loss functions (e.g., cross-entropy) respectively defined for the ground-truth labels and the teacher model M_2's outputs, and η denotes the learning rate as usual. The corresponding parameter values in M_2 are incorporated as added constraints. This way, the student model receives extra

supervision from the teacher model during training, beyond optimizing for conventional learning objectives like the cross-entropy loss subject to the ground-truth training labels.

5.1 Client: KD-enhanced parameter tuning

Observe that a DNN trained locally on an edge device encapsulates the "knowledge" extracted from the private local data. Therefore, to take full advantage of the edge data distributed across devices without exchanging the private data, our algorithm instead shares the DNN models trained independently on each device. The ensemble of DNNs from other devices serves as the "teacher" to guide the current device's model just like in standard mutual knowledge distillation.

Our algorithm proceeds as follows.

(i) Each participating endpoint device (E_i) first tunes their adapted version of DNN model (M_i) with locally available training data until convergence.

(ii) Each endpoint sends its current model along with its loss and accuracy statistics to the central coordinator and waits for a response, namely a set of models (M_1 to M_n) trained on the other devices. An operator is added over the n outputs of these models (M_1 to M_n), taking their average as the final output of the model ensemble.

(iii) KD-enhanced tuning is then invoked to optimize the parameters θ_i of model M_i: $\theta_i = \theta_i - \eta\nabla_{\theta_i}\{\phi(y, M_i(x)) + \varphi(\frac{1}{n-1}\sum_{j\neq i} M_j(x), M_i(x))\}$. Namely, the outputs of each local model M_i are compared with both the ground-truth labels y and the outputs of the assembled teacher model to calculate loss. We follow similar hyperparameter settings as in [37], using cross-entropy loss for ϕ and Kullback-Leibler (KL) divergence [45] for φ to measure the distance between the teacher and local models, and a default value 0.001 for η.

(iv) Now loop over steps (i) to (iii) until the model finally converges. Noticeably, to improve generalization and avoid being skewed by some poor performing models, we randomly skip $max(n/10, 1)$ of the models used for each round of KD-enhanced tuning in step (iii).

Privacy-aware tuning. Although less privacy-sensitive than the training data, DNN models can still leak information from the private training data. To overcome this privacy leak, we can add noise to the fine-tuning process to achieve differential privacy [39, 56]. The noise can be added to either the training data, or the model parameters sent to the *Mistify* server. However, the latter provides less privacy protection, is easier to "denoise", and does not provide fine-grained control easily.

Therefore, we augment the algorithm above with an optional step after (i). Specifically, we add Laplacian noise to the local training data, and train the model (M_i) for additional epochs until convergence. Then, this noisy model (M_i') is sent to the central coordinator in step (ii). This provides differential privacy to the model parameters and reduces information leakage from the private data. The level of noise added is chosen empirically according to existing privacy-preserving machine learning practice (e.g., PATE [56] and Myelin [39]) with the same level of privacy loss preference (e.g., $\varepsilon < 5$).

Mistify is amenable to this differentially private approach by design. As *Mistify* aims to scale to a large batch of end devices (hundreds or more), potentially there is a large number of peer models to draw from during the intermediate steps. Even if the individual noisy intermediate model (M_i') is less accurate than its noiseless counterpart, the accuracy loss is compensated for by the ensemble of other peer models [55].

5.2 Server: Client model coordination

One particular concern of our aforementioned algorithm is whether the models used for KD-enhanced tuning indeed add knowledge rather than noise. Fortunately, our approach is supported by the evidence of correlation between training data and the models. First, datasets from nearby edge devices exhibit spatio-temporal correlation [32, 33, 78]. Second, given training datasets sufficiently similar in their semantic contexts (e.g., types of objects, hidden feature occurrence frequencies), the models thus trained perform semantically equivalent functionality and can provably generalize to achieving the same capability [53, 67, 80].

In practice, we use common spatio-temporal hints (e.g., location, time, view angle) sent by each client along with their models as a coarse-grained mechanism to estimate data correlation. There are myriad alternative lightweight approaches to measure dataset similarity without piece-wise comparison of the actual raw data (e.g., by calculating dataset feature summaries [46, 57]). They are easily pluggable into *Mistify* with the corresponding API implementations. Regardless of the exact metrics used to measure correlation, they are represented as multi-dimensional vectors. The central coordinator on the *Mistify* server maintains a Locality-Sensitive Hashing (LSH [26]) structure to index the vectors for large-scale nearest neighbor lookup at a sublinear complexity [21].

6 Run-time model adaptation

Existing algorithms and libraries only port DNN models statically in a batch mode. Instead, *Mistify* further provides a *streaming* mode, where the client actively monitors runtime changes of the resource and performance constraints and requests new model generation in response to such dynamics.

Constructing a multi-branch model. To support on-the-fly adaptation to fluctuating resource constraints, each DNN model is further constructed in a multi-branch form (Figure 5) during the architecture adaptation process. First, the aforementioned adaptation algorithm is triggered as usual until the constraints specified in the configuration are satisfied. Now, besides continuing to adapt to other configurations, a new adaptation thread is spawned. This thread separately adapts the current DNN into a k-branch DNN. For instance, a 5-branch DNN is built by freezing the first few layers and adapting the remaining layers towards 5 different configurations, whose resource budgets range from $\frac{1}{3}$ of to 3 times that of the original DNN model. The branches share the same base,

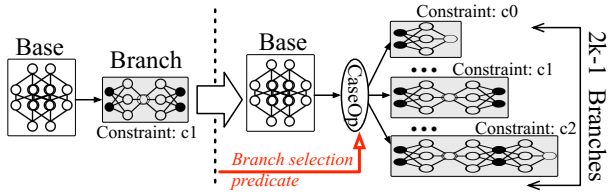


Figure 5: Multi-branch model construction.

achieve the same inference task, but satisfy different resource budgets and performance goals. In practice, k is set based on the extent of fluctuation typically observed in the resource availability and performance targets. After fine-tuning the parameters of the multi-branch DNN, we add a case conditional operator (e.g., `tf.case` for Tensorflow) between the base and different branches.

Foreground path: downtime free branch switching. The foreground path is tightly coupled with the user-facing inference serving logic, performing real-time adjustments of the current DNN model based on the dynamic constraints. To achieve this, *Mistify* picks a branch from the multi-branch DNN with the closest resource and performance profiles. The branch switching is done on the fly by setting the corresponding value of the conditional variable (the red arrow in Figure 5) of the case operator in the DNN, avoiding additional overhead such as memory allocation and runtime resource instantiation.

Background path. Meanwhile, in the background, the *Mistify* client will send the new adaptation configuration to the *Mistify* server. The server compares the new configuration with existing ones in terms of their resource constraints, based on the partial ordering explained in Section 4, in order to retrieve the immediate predecessor configuration of this new incoming one. Then, the new DNN model is incrementally adapted from the corresponding "predecessor" DNN, until the constraints of the new configuration are met.

7 Implementation

We implement *Mistify* on TensorFlow (TF) 1.13 [13] (Figure 2), consisting of around 8.5K lines of Python code for both the server and client modules. The source code will be available later at [8].

Interfacing with the native environment. Recall that *Mistify* can be activated at two stages (Section 3), when initializing inference serving or during the run time. For the former, the function `tfhub.load()` is intercepted to trigger the model porting process (when fed the special argument). For the latter case, the *Mistify* runtime monitor is by default registered with the live `Session` of the TensorFlow engine to collect runtime statistics (`tf.RunMetadata`), and invoke the *Mistify* client to initiate the re-adaptation process on demand. The foreground branch switching is implemented by assigning a suitable value to the predicate variable of the `tf.case` op.

Encapsulating adaptation algorithms. *Mistify* implements wrappers over two representative, state-of-the-art adaptation algorithms, MorphNet [31] (using sparsifying regularizers) and ChamNet [25] (using evolutionary algorithms). Adding new adaptation algorithms to *Mistify* is fairly easy, following the process outlined in the MorphNet case study in Section 4.2. Each wrapper implementation around these algorithms for *Mistify* requires around 100 lines of code (LoC), which is fairly modest compared to the thousands of LoC in the original codebases of these algorithms.

8 Evaluation

Hardware setup. Following Figure 2, a Linux server with 8-core 2.1 GHz Intel Xeon CPU, and NVIDIA 2070 GPU acts as the server side of *Mistify*. For the client-side operations of *Mistify*, we use a server with a low-end NVIDIA P600 GPU, a Google Edge TPU [4], and a Samsung S9 smartphone, to represent diverse types of edge hardware.

Application benchmarks. Computer Vision (CV) and Natural Language Processing (NLP) tasks almost dominate deep learning use scenarios. We select one workload each, *Object Recognition* and *Question & Answering* corresponding to the two application categories, as the representative benchmarks. While there are numerous other CV and NLP applications, for example, *scene segmentation* for CV and *machine translation* for NLP, these are based on DNN models derived from the same base structures as those used for our benchmarks (e.g., ResNet blocks for *object recognition* and *detection*, Transformer blocks for *Q&A* and *named entity recognition*). Therefore, the results obtained for our benchmarks are representative of a wide range of scenarios.

Specifically, we select three carefully designed, state-of-the-art DNNs, MobileNet [38], ResNet50 [35], and ResNeXt101 [51], with increasing computation complexity (0.5 to 16 GFLOPs), parameter size (16 to 320 MB), and accuracy (68% to 79%) for *object recognition*. MobileNet is originally designed for mobile devices, whereas the other two mostly run in the cloud. For *Q&A*, the input is a question along with a context paragraph containing the answer to the question. The "accuracy" metric for this is the Exact Match (EM) score, i.e., whether the output answer exactly matches the question. We prepare two DNNs, BiDAF [68], and BERT [28]. The former is lightweight but task-specific (customized for *Q&A*) (10× MB), whereas BERT is much larger, generically supporting various downstream tasks.

Datasets. We use domain specific standard datasets to adapt network architectures, fine-tune their parameters, and validate their performance. Specifically, ImageNet [27] and Cifar100 [44] are used for *object recognition*, whereas SQuADv1.1 [61] is used for *Q&A*.

8.1 Collective architecture adaptation

Collective adaptation time. We generate 128 different adaptation configurations based on four DNNs (MobileNet, ResNet50, ResNeXt101, and BERT). Among these, the least

Table 2: Accuracy - collectively adapted models (*Mistify*) vs individually adapted models (Per-case)

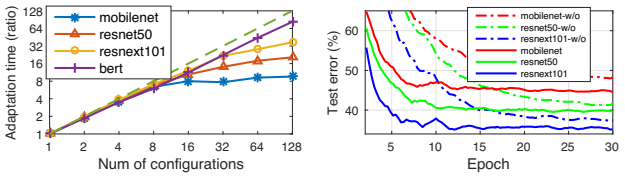
DNN	Per-case (%)	*Mistify* (%)	Relative diff (%)
MobileNet	55.8	54.7	-2.0%
	69.4	69.5	+0.1%
ResNet50	68.2	68.0	-0.3%
	72.9	72.5	-0.6%
ResNeXt101	74.0	74.3	+0.4%
	77.6	77.9	+0.3%
BERT	71.4	70.6	-1.2%
	79.1	78.8	-0.4%

and most demanding configurations respectively constrain the adapted DNNs to 2× and 0.5× the default DNN memory usage and computation complexity. Then, we select different subsets of these 128 configurations, adapt all of them with and without *Mistify*, and compare their overall time consumption to evaluate our collective adaptation approach (Section 4.3). Figure 6(a) shows the relative time needed without *Mistify* over with *Mistify*. *Mistify* accelerates the overall adaptation time almost linearly with the number of configurations when it is less than 10, consistently achieving around 10x acceleration even for DNNs as small as MobileNet. For large DNNs, such as BERT, that are structurally more amenable to adaptation (i.e., easier to prune a subset of the network without affecting validation accuracy), the acceleration scales well with over 100 configurations.

Adaptation quality. Next, we examine the quality of the DNNs collectively adapted by *Mistify* versus those adapted individually. Table 2 shows two rows for each network, corresponding to compression and expansion by a factor 4 with respect to the complexity and memory consumption of the original DNN. This spans the range from low- to high-end hardware [75]. For instance, the inference times of the compressed and expanded ResNet50, running on a Google Nexus 5 (low-end, 2013 model) and a Samsung Galaxy 10 (high-end, 2019 model), are both around 30 ms, low enough for practical usage. "Accuracy" corresponds to the EM score (exactly matching the ground-truth answer) for NLP. To avoid being affected by the parameter tuning quality, all adapted DNNs are trained with the whole datasets, and without considering any device-specific constraints. *Mistify*'s collective adaptation achieves almost the same accuracy compared to the case-by-case strategy, with less than 0.5% accuracy loss for most cases and only 1% for the worst scenario (e.g., when adaptation configurations are incompatible with total ordering, causing the overall adaptation path to detour substantially). These are within the typical range of accuracy loss in exchange for resource efficiency [64].

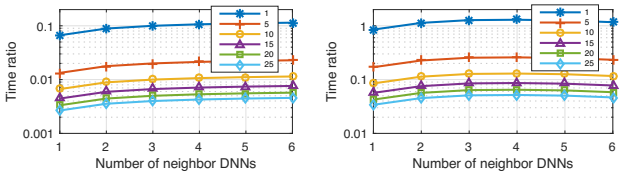
8.2 Parameter tuning

We use a more specialized dataset Cifar100 to evaluate parameter tuning on the edge. The whole dataset is partitioned into subsets, mimicking the local data of each edge device.



(a) DNN apdatation time. (b) Parameter tuning time.

Figure 6: Speed and performance improvements for architecture adaptation and parameter tuning with *Mistify*.



(a) MobileNet (b) BERT

Figure 7: The ratio of communication time over training time, reflecting the scalability of fine-tuning in *Mistify*.

Convergence speed and quality. We compare the convergence speed and test accuracy for three different networks (MobileNet, ResNet50, and ResNeXt101), with and without *Mistify*'s support for parameter tuning (Section 5.1). Figure 6(b) shows that, even without additional data, KD-enhanced parameter tuning (solid lines) already achieves over 3× faster convergence as well as better accuracy.

Scalability. We assess the scalability of the parameter tuning algorithm (Section 5.1) in terms of the ratio of the communication time over the training time given different network bandwidths. We consider two model extremes, MobileNet (very compact) and BERT (very sophisticated). In Figure 7, each line corresponds to a specific network bandwidth in MB/s. When the network bandwidth exceeds 5 MB/s, our algorithm is consistently scalable, with communication merely taking less than 15% of the time relative to training. Further, the lines almost flatten when more than three neighbors' DNNs are used, so using more peer DNNs for our tuning does not impact scalability.

Accuracy of parameter tuning. We randomly partition Cifar100 and SQuAD each into 5 subsets, each used by an edge device for local training. Then, we compare the fine-tuning accuracy using different approaches. Table 3 shows that knowledge distillation (KD) improves parameter tuning accuracy by 40% over local training alone. Compared to the ideal distillation case where an exceptionally accurate teacher network is available (a pre-trained, cloud version), the ensemble of 4 peer DNNs achieves within 10% of the optimal KD, despite using half the training data and adding differential privacy to the model parameters.

8.3 Run-time model re-adaptation overhead

The foreground path. Switching DNNs in response to the run-time dynamics (Section 6) incurs two types of overhead:

Table 3: The accuracy of parameter tuning with *Mistify*.

Scenario	DNNs (%)		
	MobileNet	ResNet50	BERT
Local training	39.7	43.9	22.5
KD	66.4	75.3	78.8
1-peer tuning	53.8	61.5	51.9
2-peer tuning	58.1	67.2	65.6
4-peer tuning	59.8	69.0	71.8

Table 4: Additional number of parameters and DNN switching time overhead.

DNN	Addi. / orig. params (M)	Time (s)
MobileNet-b3	2.67 / 3.43	2.11
MobileNet-b5	4.57 / 3.43	
ResNet50-b3	18.2 / 23.9	3.34
ResNet50-b5	31.7 / 23.9	
BERT-b3	92.4 / 110	21.84
BERT-b5	171 / 110	

i) additional memory to store alternate DNNs; and/or ii) downtime for loading the new DNN and on-demand preparation of the resource runtime. The model size corresponds to the in-memory size, not the on-disk size of the serialized form. Table 4 illustrates the trends of additional memory or time consumption for different DNNs. The suffix "-bk" means adding k branches to the adapted DNN. Holding 75% to 1.5× more parameters in memory is an affordable cost for modern hardware, but can save us 2 to 20 seconds by avoiding loading new DNNs on the critical path of inference serving.

Latency of modifying the configuration tree. We also measure the latency of generating a configuration tree (Section 4.3) given various numbers of configurations. Using 1000 configurations, each consisting of 4 constraints, the whole tree is built in 37 ms, and inserting into an existing configuration tree only takes microseconds.

8.4 End-to-end performance

Based on the device specifications in Table 1 and typical latency requirements for vision and NLP tasks [1, 63], we generate different combinations of memory, complexity, and latency constraints as the execution settings, and evaluate the quality of the DNN models tailored by *Mistify*. We further compare the manual overhead involved in *Mistify* with running MorphNet [31] and ChamNet [25] directly.

Balancing performance and resource usage. We set the memory budget to range from 0.1 GB to 10 GB, covering embedded IoT devices to edge servers. The computation complexity constraints for running inference on a DNN vary between 0.1 to 100× GFLOPs, roughly equivalent to achieving 10s of microseconds of inference latency for resource-constrained devices and powerful edge servers alike.

Figure 8 shows the three-way trade-offs between accuracy, latency, and resource consumption. The top three plots correspond to *recognition*, the lower three to *Q&A*. *Mistify* reduces the compute requirements by over 20× with less than 5% accuracy loss for the CV workloads, and could achieve 50× reduction of complexity in exchange for 12% relative quality-of-result degradation. Note that the accuracy loss is due to the adaptation algorithms, not *Mistify* itself. Similarly, *Mistify* consistently achieves a near-optimal and practically usable accuracy (comparable to existing hand-tuned on-device models in production [25, 49]) with between 0.5 to 10 GB of run-time memory usage, hence significantly decreasing the deployment complexity for state-of-the-art DNN models on the edge. Mapping resource consumption to inference time, *Mistify* consistently achieves near-optimal accuracy performance even when the latency requirements vary by 8 to 10×, corresponding to using accelerator hardware ranging from advanced, datacenter grade to low-power, lightweight devices.

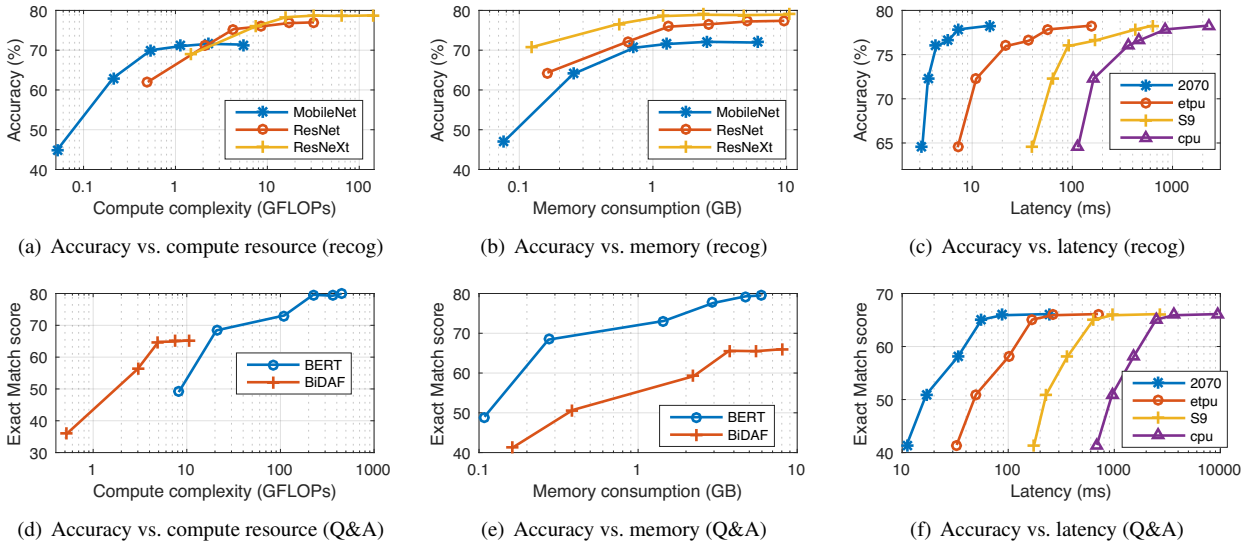
Reducing manual overhead. We further assess the end-to-end manual effort and time overhead needed to port a pre-designed DNN to different edge devices. The manual overhead is quantified with two metrics: lines of code (LoC) needed for code addition or modification, and number of files (NoF) touched. The former depicts the overall overhead, and the latter one captures the scatteredness of the modifications, which correlates with the probability of making mistakes. For NoF, we follow a typical file organization [12], i.e., model definition, training, evaluation, and other stages are separated into different files or folders.

Table 5 demonstrates that *Mistify* reduces the overall modification needed in LoC by 7 to 10×. More importantly, *Mistify* exposes high-level configuration files to users, obviating the need for source script modifications. *Mistify* only requires editing one file. Thus, it can reduce the number of files users need to access by orders of magnitude (over 100×). Finally, *Mistify* can batch-adapt to 100 execution settings using less than 3% of the time needed for the other approaches, highlighting the enormous potential of harnessing the correlation among configurations to optimize the overall porting efficiency.

9 Related work

We are not aware of any prior work that aims at providing an automatic porting *service* bridging DNN design and seamless edge deployment. The most related work revolves around model adaptation and knowledge distillation *algorithms*.

Model adaptation. Production DNN models hand-tuned by experts can run fast and accurately on mobile devices [38, 40, 49, 62]. The essential techniques include quantization, sparsification, and neural block optimization. Recently, Distiller [11], AMC [36], MorphNet [31], OFA [22], ChamNet [25], and many neural architecture search (NAS) works [17, 50, 65, 83, 84] systematically explore the search space for the optimal neural network structure, obviating the hand-tuning by experienced experts. However, none of them is directly usable like *Mistify*, because all are still *algorithms*, requiring manually annotating source code to construct the



(a) Accuracy vs. compute resource (recog)
(b) Accuracy vs. memory (recog)
(c) Accuracy vs. latency (recog)
(d) Accuracy vs. compute resource (Q&A)
(e) Accuracy vs. memory (Q&A)
(f) Accuracy vs. latency (Q&A)

Figure 8: The dynamic tradeoff between latency, accuracy, and resource consumption with *Mistify*.

Table 5: Comparison of overhead for porting DNN to edge with/without *Mistify*.

Metrics	2 configurations				10 configurations				100 configurations			
	Manual	Mor.	Chm.	*Mistify*	Manual	Mor.	Chm.	*Mistify*	Manual	Mor.	Chm.	*Mistify*
Lines of Code	>0.2k	55	97	**6**	>1k	138	159	**14**	>10k	782	511	**104**
Num of Files	6	4	5	**1**	30	12	32	**1**	300	102	302	**1**
Total time (%)	100			**54.2**	100			**12.5**	100			**2.86**

adaptation logic, hence not scalable to edge scenarios with multiple adaptation instances. None supports on-device fine-tuning or considers run-time adjustments. *Mistify* is orthogonal as an automated *system framework* and can incorporate them as pluggable algorithmic modules.

While frameworks like TF-Lite [2], PyTorch [58], and MCDNN [34] provide some model *compression* and *switching* support, *Mistify* differs in the techniques supported and the level of manual efforts needed. To generate a good model, careful model architecture design is essential, which normally requires significant expertise. *Mistify* abstracts away the model architecture searching process with the configurable APIs to make it accessible to non-experts, automating the end-to-end process and optimizing for batch model generation.

Knowledge distillation. Initially proposed as an optimization for model training, *knowledge distillation* transfers "knowledge" (i.e., parameter values) from a *teacher* network to a *student* network [37]. The idea is then extended to *mutual distillation* among peer models [15, 47, 81]. *Mistify* adopts and revises the general idea in a *selective distillation* manner to improve edge training accuracy while enhancing privacy.

Edge-centric deep learning inference engines. Emerging frameworks such as TF-Lite [2] and more [7, 29, 54] are optimized for inference serving on mobile and IoT devices, aiming to hide the deployment complexity from developers and device users. However, the interface exposed by existing engines only permits model download from the cloud (or the central server), without tailoring to edge runtime requirements and constraints, proactively or reactively. In contrast, *Mistify* provides an interface for two-way state exchange and a feedback loop between the cloud and the edge, facilitating targeted model design and efficient execution on the edge.

10 Conclusion

Deep learning models today are typically trained in the cloud and then ported to edge devices manually. Not only is manual porting unscalable, it indicates a lack of separation between model design (optimized for accuracy) and deployment (optimized for resource efficiency).

In this paper, we design and implement *Mistify*, a framework to automate this porting process, which reduces the DNN porting time needed to cater to a wide spectrum of edge deployment scenarios by over 10×, incurring orders of magnitude less manual effort. *Mistify* not only provides a useful service to complete the transition from DL workload design to deployment on the edge, but cleanly separates these two stages. We believe the system will further facilitate advanced model design and seamless model deployment.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Eric Schkufza, for their insightful comments. Julia McClellan helped with early exploration of the work. This work is supported by the National Science Foundation under Grant No. CNS-1815115 and a Google Faculty Research Award.

References

[1] Aetina: dedicated AI computing solution at edge. https://www.aetina.com/index.php.

[2] Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite.

[3] Driving intelligent retail with AI. https://www.nvidia.com/en-us/industries/retail/.

[4] Edge tpu: Google's purpose-built asic designed to run inference at the edge. https://cloud.google.com/edge-tpu.

[5] Foghorn: The Original Edge-Native AI Platform. https://www.foghorn.io/edge-ai-platform/.

[6] Gartner highlights 10 uses for ai-powered devices. https://www.gartner.com/en/newsroom/press-releases/2018-03-20-10-uses-for-ai-powered-devices.

[7] Integrate machine learning models into your app. https://developer.apple.com/documentation/coreml.

[8] Mistify github repo. https://github.com/PatrickGuo/Mistify.

[9] MobiSys'19 IoT Day. https://www.sigmobile.org/mobisys/2019/iot_day_program/.

[10] MorphNet: a library of learning deep network structure during training. https://github.com/google-research/morph-net.

[11] Neural Network Distiller by Intel AI Lab: a Python package for neural network compression research. https://github.com/NervanaSystems/distiller.

[12] TensorFlow Official Models. https://github.com/tensorflow/models.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[14] J. M. Alvarez and M. Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278, 2016.

[15] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[16] Avnet. Ai at the edge: The next frontier of the internet of things, 2018.

[17] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[18] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[19] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[20] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[21] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[22] H. Cai, C. Gan, and S. Han. Once for all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[23] N. Chen, Y. Chen, Y. You, H. Ling, P. Liang, and R. Zimmermann. Dynamic urban surveillance video stream processing using fog computing. In *2016 IEEE second international conference on multimedia big data (BigMM)*, pages 105–112. IEEE, 2016.

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[25] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.

[26] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[29] M. Dukhan, Y. Wu, and H. Lu. Qnnpack: open source library for optimized mobile deep learning.

[30] D. Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[31] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[32] P. Guo, B. Hu, R. Li, and W. Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 19–34, 2018.

[33] P. Guo and W. Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, 2018.

[34] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136, 2016.

[35] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[36] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[37] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[39] N. Hynes, R. Cheng, and D. Song. Efficient deep learning on multi-source private data. *CoRR*, abs/1807.06689, 2018.

[40] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[41] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14. ACM, 2019.

[42] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[43] N. V. Kim and M. A. Chervonenkis. Situation control of unmanned aerial vehicles for road traffic monitoring. *Modern Applied Science*, 9(5):1, 2015.

[44] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 and cifar-100 datasets. *URl: https://www. cs. toronto. edu/kriz/cifar. html*, 6, 2009.

[45] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[46] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther. Autoencoding beyond pixels using a learned similarity metric. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1558–1566. JMLR.org, 2016.

[47] T. Li, J. Li, Z. Liu, and C. Zhang. Knowledge distillation from few samples. *arXiv preprint arXiv:1812.01839*, 2018.

[48] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 255–266. IEEE Press, 2016.

[49] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[50] H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[51] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 181–196, 2018.

[52] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[53] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586, 2018.

[54] S. Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 2019.

[55] N. Papernot, M. Abadi, Ú. Erlingsson, I. J. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[56] N. Papernot, S. Song, I. Mironov, A. Raghunathan, K. Talwar, and Úlfar Erlingsson. Scalable private learning with pate. In *ICLR*, 2018.

[57] S. Parthasarathy and M. Ogihara. Exploiting dataset similarity for distributed mining. In *International Parallel and Distributed Processing Symposium*, pages 399–406. Springer, 2000.

[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[59] V. Popov, M. Kudinov, I. Piontkovskaya, P. Vytovtov, and A. Nevidomsky. Distributed fine-tuning of language models on private data. 2018.

[60] W. Qualcomm. We are making on-device ai ubiquitous, 2018.

[61] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, 2016.

[62] S. Ravi. Projectionnet: Learning efficient on-device deep networks using neural projections. *arXiv preprint arXiv:1708.00630*, 2017.

[63] S. Ravi and Z. Kozareva. Self-governing neural networks for on-device short text classification. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 887–893, 2018.

[64] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[65] E. Real, C. Liang, D. R. So, and Q. V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 2020.

[66] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[67] A. Ruoss, M. Baader, M. Balunović, and M. Vechev. Efficient certification of spatial robustness. *arXiv preprint arXiv:2009.09318*, 2020.

[68] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[69] A. Shin, D. Y. Kim, J. S. Jeong, and B.-G. Chun. Hippo: Taming hyper-parameter optimization of deep learning with stage trees. *arXiv preprint arXiv:2006.11972*, 2020.

[70] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.

[71] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.

[72] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[73] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[74] J. Wang, J. Pan, and F. Esposito. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things*, page 7. ACM, 2017.

[75] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[76] C. Xia, J. Zhao, H. Cui, X. Feng, and J. Xue. Dnntune: Automatic benchmarking dnn models for mobile-cloud computing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.

[77] Y. Xiong, R. Mehta, and V. Singh. Resource constrained neural network architecture search: Will a submodularity assumption help? In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1901–1910, 2019.

[78] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144, 2018.

[79] J. Yim, D. Joo, J. Bae, and J. Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4133–4141, 2017.

[80] J. Zhang, Y. Wang, P. Molino, L. Li, and D. S. Ebert. Manifold: A model-agnostic framework for interpretation and diagnosis of machine learning models. *IEEE transactions on visualization and computer graphics*, 25(1):364–373, 2018.

[81] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu. Deep mutual learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4320–4328, 2018.

[82] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. 2017.

[83] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[84] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8697–8710. IEEE Computer Society, 2018.