Tiny Updater: Towards Efficient Neural Network-Driven Software Updating

Linfeng Zhang, Kaisheng Ma*
Institute for Interdisciplinary Information Sciences, Tsinghua University
zhanglinfeng1997@outlook.com, kaisheng@mail.tsinghua.edu.cn

Abstract

Significant advancements have been accomplished with deep neural networks in diverse visual tasks, which have substantially elevated their deployment in edge device software. However, during the update of neural network-based software, users are required to download all the parameters of the neural network anew, which harms the user experience. Motivated by previous progress in model compression, we propose a novel training methodology named Tiny Updater to address this issue. Specifically, by adopting the variant of pruning and knowledge distillation methods, Tiny Updater can update the neural network-based software by only downloading a few parameters (10%~20%) instead of all the parameters in the neural network. Experiments on eleven datasets of three tasks, including image classification, image-to-image translation, and video recognition have demonstrated its effectiveness. Codes have been released in https://github.com/ArchipLab-LinfengZhang/TinyUpdater.

1. Introduction

With the availability of large-scale datasets [15, 16, 51] and high-performance computing platforms, deep neural networks have achieved remarkable achievements in various visual tasks such as image classification [18, 28, 75, 80], segmentation [53, 62], and object detection [50, 67, 68, 76]. Encouraged by their impressive performance, numerous software developers have effectively integrated neural networks into their software products and deployed them on edge devices such as mobile phones and tablet computers.

Typically, the development roadmap for a neural network-based software follows the paradigm illustrate in Figure 1(a). Initially, users install the software by downloading all the neural network parameters from a cloud platform. Subsequently, as the software interacts with its users, an abundance of new training data and requirements can be collected. Then, software developers may retrain the neural network with the gathered data and update their software to enhance its performance. However, during the retraining phase, all the neural network parameters are typically changed compared to their values before updating. Consequently, users are compelled to download all the parameters of the neural network again from the cloud platform, seriously impairing their experience. While recent research has explored text prompts and adapter layers to fine-tune a large-scale pre-trained model at a low training cost [52, 79], there have been no prior efforts to reduce the download cost during model updating, which has a more direct impact on users with edge devices.

This paper proposes the challenge of efficient model updating with the objective of reducing the download overhead of neural network-based software during updating. As
To tackle this challenge, we propose a novel neural network training framework named Tiny Updater. Motivated by previous works in model compression, Tiny Updater is composed of the variants of two typical model compression techniques - neural network pruning and knowledge distillation. Firstly, to determine which channels or layers in neural networks should be modified during updating, Tiny Updater applies the pruning technique. As shown in Figure 2, it iteratively calculates the $L_1$-norm distance between the pre-updating model and the post-updating model parameters. The channels with smaller distances are deemed unnecessary for updating and are pruned to their original values before updating. Conversely, the channels with larger distances are considered essential for updating, and thus they are actually changed. Secondly, during the retraining period, we propose to improve the performance of the partially-updated model by distilling the knowledge from a fully-updated teacher model. The partially-updated student model is trained to give predictions that are similar to those of the fully-updated teacher model by optimizing the knowledge distillation loss. This ensures that the partially-updated model achieves comparable performance with the fully-updated model.

Extensive experiments have been conducted on three datasets and three different tasks, including, image classification, video classification, and image-to-image translation, using seven different neural networks. Experimental results demonstrate that Tiny Updater can update the model by changing only 10% to 20% of the parameters with minimal performance degradation. In summary, the main contributions of this paper can be outlined as follows.

- To the best of our knowledge, we first propose the challenge of efficient model updating to reduce the communication cost of downloading neural network parameters to the edge devices for software updating.
- To tackle this challenge, we propose Tiny Updater, which prunes the fully-updated model to the pre-updating model and distills the knowledge from the fully-updated model to the partially-updated model.
- Extensive experiments have been conducted on three tasks with eleven datasets, including image classification, image-to-image translation and video recognition. Experimental results demonstrate that Tiny Updater can reduce the training overhead by 80%–90% with almost no performance degradation.

2. Related Work

2.1. Neural Network Pruning

Neural network pruning is one of the most effective methods for deep neural network compression by deleting the unimportant neurons [42, 47], filters [29, 38], channels [30, 90] and layers [8, 69]. LeCun et al. and Hassibi et al. first propose to prune neural networks depending on the Hessian matrix [27, 41]. Recently, Han et al. propose deep compression, which finds the important connection in neural networks with its absolute value and then iteratively prunes them [24]. Liu et al. train a meta-network to generate the weights of neural networks and then apply it to search the best-pruned architecture [38]. Ding et al. propose to apply a LSTM to learn the hierarchical characteristics of deep neural networks and then generates the corresponding pruning scheme [17]. Liu et al. propose joint multi-dimension pruning to prune the channels, layers and resolutions at the same time [59]. Frankle et al. propose the lottery ticket hypothesis and find that a standard pruning technique naturally uncovers subnetworks whose initializations made them capable of training effectively [21]. Besides classification, neural network pruning has also be utilized in image-to-image translation [37, 42], unconditional image generation [56], object detection [22, 66], video retrieval [10] and pre-trained language models [14].

In this paper, Tiny Updater introduces neural network pruning methods to find the parameters which are actually valuable for model updating. Its main difference compared with previous pruning methods is that Tiny Updater prunes the parameters to their value before updating instead of zero, which makes it easier to be solved than traditional pruning. Our experiments also show that Tiny Updater can be utilized on the pruned models.

2.2. Knowledge Distillation

Knowledge distillation, also known as student-teacher learning, has become one of the most popular deep learning techniques in various domains [6, 49, 72, 85, 86]. It firstly trains an over-parameterized teacher model and then distills teacher knowledge to a lightweight student. By training the student to mimic the prediction results of the teacher, the student can inherit the knowledge learned by the teacher. Thus, it can achieve better performance than traditional training. The concept of knowledge distillation is first proposed by Bucliu et al. [7] to compress ensemble
of neural networks, and then extended by Hinton et al. with a temperature hyper-parameter in the softmax function [32]. Following their success, abundant methods have been proposed to distill teacher knowledge in their backbone features [42, 70], spatial attention [20, 84], task-oriented information [87], pixel-wise relation [20, 44, 49, 86], sample-wise relation [63, 65, 78], prediction residual [43] and so on.

Previous knowledge distillation methods usually transfer teacher knowledge in the categorical probability distribution. Then, abundant methods have been introduced to distill the knowledge in the teacher feature [70] and its variants, such as attention [84, 86], relational information [63, 78], self-supervised knowledge [81] and task-oriented information [87]. Besides distilling knowledge from a large teacher to a tiny student, there have also been fruitful knowledge distillation methods that distill knowledge from deeper layers to shallow layers [88, 89], from all the channels to partial channels [83], from ensemble classifiers to single classifier [40], from multiple frames to few frames [5], from RGB images to RGB-D images [23]. In this paper, we introduce knowledge distillation by distilling the knowledge from a teacher model, which has all the parameters changed during updating to a student model, which is only updated by changing a few parameters.

2.3. Incremental Learning

Inspired by the observation that human beings can incrementally learn knowledge about new tasks and categories without forgetting the old knowledge, incremental learning (a.k.a. lifelong learning) is proposed to give the neural networks the similar ability. Elastic weight consolidation (EWC) is proposed to maintain the knowledge of old tasks and categories by first estimating the importance of each neuron with Bays estimation or Fisher information matrix and then training them with a consolidation constraint [39, 55, 77]. Then, memory aware synapse [1] is proposed to compute the importance score of neural network parameters in an unsupervised and online manner, which is also similar to the Hebbian learning in biological systems [31]. Besides these consolidation-based methods, knowledge distillation methods have also been utilized in incremental learning to maintain the old knowledge by learning its response to new tasks [13, 48, 92].

Both the proposed efficient model updating and incremental learning expect the neural network to maintain its knowledge of old parameters. Their main difference is that:

(a) Incremental learning does not limit the change in old parameters strictly, while efficient model updating has a direct constraint on the number of changed parameters to reduce the download overhead for edge devices.

(b) Incremental learning usually assumes that the data for old knowledge is not available while efficient model updating can still access data, which is more practical to industrial applications.

(c) The target of incremental learning is to build human-like artificial intelligence which can incrementally learn various tasks, while the target of Efficient Model Updating is to reduce the number of parameters updated when new data for the same task is collected.

2.4. Efficient Pretrained Model Finetuning

Large-scale pretrained models recently have shown powerful representation ability in both natural language processing and computer vision. However, since these pretrained models usually have billions of parameters, directly finetuning them on downstream tasks usually suffers from massive training overhead, making them impractical in real-world applications. Recently, prompts methods have been proposed to address this problem by designing specific input templates for the specific downstream tasks [4, 25, 45, 52, 54, 79, 91]. Moreover, adapter methods have also been intro-
duced to finetune several additional trainable layers instead of the whole pre-trained model [3, 33]. These methods are firstly proposed for pre-trained language models and then extended to language-vision multi-modal models and vision models [12, 36, 82]. Both these efficient finetuning methods and the proposed efficient model updating aim to freeze the parameters of neural networks. Their main difference is that: (a) Efficient finetuning is usually applied to the models which are firstly pre-trained on large-scale datasets to learn task-unbiased knowledge and then finetuned for specific downstream tasks. Instead, the pre-updating models in Efficient Model Updating are trained with very few training samples (e.g. only 20% in our experiments) and then updated with the new data collected from users. (b) The target of efficient finetuning is to reduce the training overhead of pre-trained models and improve their performance in downstream tasks. In contrast, the target of Efficient Model Updating is to reduce the download overhead during software updating. (c) Besides, adapter-based methods can be considered as a special case in Tiny Updater where all the parameters of backbone layers are frozen and all the adapter layers are updated. Our experiments also demonstrate that Tiny Updater achieves clearly better performance than directly applying efficient model updating methods. Please refer to Appendix A for a more detailed comparison among the proposed efficient model updating, incremental learning, and efficient pretrained model finetuning.

3. Methodology

3.1. Efficient Model Updating

Given a training dataset \( \mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\} \), the software developers firstly train a deep neural network \( \mathcal{F} \) with parameter \( \Theta \) for their software. After deploying \( \mathcal{F} \) on the edge devices, abundant new training samples \( \{(x_{n+1}, y_{n+1}), \ldots, (x_{n+m}, y_{n+m})\} \) can be collected from users and the dataset can be extended as \( \mathcal{D}^+ = \mathcal{D} \cup \{(x_{n+1}, y_{n+1}), \ldots, (x_{n+m}, y_{n+m})\} \). Then, software developers can re-train \( \mathcal{F} \) on \( \mathcal{D}^+ \) to obtain better performance, whose parameters can be denoted as \( \Phi \). Usually, during this re-training step, all the values in \( \theta \) can be totally different from that in \( \Phi \). Thus, the users have to download all the parameters of \( \Phi \) to update the software, which harms the user experience and limits the frequency of updating neural-network-driven software. In such kind of paradigm, we name the model before updating as the pre-updating model, the model after updating as the post-updating model, the model with all the parameters updated as the fully-updated model, and the model with partial parameters updated as the partially updated model. The target of efficient model updating is to obtain a partially updated model which has most of the parameters unchanged while achieving similar performance to the fully-updated model.

### Algorithm 1 The proposed Tiny Updater.

**Input:** Dataset \( \mathcal{D}^+ = \{(x_1, y_1), \ldots, (x_{n+m}, y_{n+m})\} \), the pre-updating model \( \mathcal{F} \) with parameter \( \Theta \), an expected ratio of updated parameters \( \tau \).

**Output:** The partially-updated model \( \mathcal{F} \) with parameter \( \Phi \).

// The partially-updated model \( \mathcal{F} \) with parameter \( \Phi \).

**Initialize** the partially-updated model \( \mathcal{F} \) with parameter \( \Phi \).

while \( \mathcal{F}_\Phi \) is not converged do:

- Sample a batch of data \( \mathcal{X} \) from \( \mathcal{D}^+ \)
- Compute \( \hat{y} := \mathcal{F}_\Phi(\mathcal{X}) \)
- Compute the task loss between \( y \) and \( \hat{y} \).
- Back propagate gradients and update \( \Phi \).

// Pruning, and re-training with KD loss.

**Initialize** the parameters of the teacher \( \mathcal{T} := \Phi \), an index set of pruned weights as \( \mathcal{I} = \emptyset \).

while \( \frac{\text{Card}(\mathcal{I})}{\text{Card}(\Phi)} < 1 - \tau \) do:

- Compute the \( L_1 \)-norm distance between \( \Phi \) and \( \theta \).
- Append the indices of channels which have relatively smaller distance to \( \mathcal{I} \).
- \( \Phi[\mathcal{I}] := \Theta[\mathcal{I}] \) // Prune \( \Phi \) to \( \Theta \).

**Return** The partially-updated model \( \mathcal{F} \) with parameter \( \Phi \).

3.2. Tiny Updater

Fruitful previous works in model compression have successfully proven that even a very tiny neural network can have powerful representation ability, which motivates us to propose to learn the knowledge in the collected training data \( \mathcal{D}^+ \) with only a few parameters instead of all the parameters. The optimization objective of Tiny Updater can be formulated as

\[
\arg\min_{\Phi} \frac{1}{n + m} \sum_{i=1}^{n+m} \mathcal{L}_{\text{task}}(x_i, y_i) \\
\text{subject to } |\Theta - \Phi|_0 < \tau
\]

where \( \mathcal{L}_{\text{task}} \) indicates the original task-specific loss function, such as cross-entropy loss for image classification. \(|\cdot|_0 \) indicates the \( L_0 \)-norm, which measures the number of non-zero elements in a tensor. \( \text{Card}(\cdot) \) denotes the cardinality, which describes the number of parameters in a tensor. \( \tau \) is a ratio threshold that determines how many parameters should be changed during updating. It is observed that when \( \theta \) in Equation (1) becomes zero, the optimization objective is similar to another deep learning technique - network prun-
Figure 3. Experiments on four fine-grained image classification datasets with ResNet50. The pre-updating and the post-updating models are trained with 25% and 100% training data, respectively. In subfigure (a), the newly collected images come from all categories uniformly. In subfigure (b), the newly collected images come from only the categories that are not available before updating.

4. Experiment

4.1. Image Classification

Experiment Setting Our method has mainly been evaluated on seven image classification datasets, including CIFAR10, CIFAR100, and ImageNet for general image classification, Stanford Car, Oxford Flower, Stanford Dog, FGVC Aircraft for fine-grained image classification. Please refer to Appendix B for the details of these datasets. Note that on the fine-grained image classification datasets, models are initialized with backbone weights pre-trained on ImageNet. In ImageNet experiments, we adopt the basic training policy from PyTorch [64] with task-oriented feature distillation [87]. On the other datasets, each model is trained by 200 epochs with the naive logit and feature knowledge distillation loss [32, 70].

Main Results The performance of Tiny Updater on ImageNet, CIFAR10, CIFAR100, and four fine-grained image classifi-
mental settings, where the pre-updating models are trained. The effectiveness of Tiny Updater in the categorical incremental updating leads to consistently higher accuracy (around 6%) compared to the random pruning, Tiny Updater with knowledge distillation can still achieve significant performance by only changing 10% parameters, indicating that Tiny Updater is also effective on the categorical incremental updating setting.

Influence from Pre-updating Model In previous experiments, the pre-updating models are trained with 25% data. With less training data, the pre-updating models tend to have less representative ability, and thus the performance of Tiny Updater tends to be reduced. In this subsection, we study how the performance of the pre-updating model influences Tiny Updater on CIFAR10 in Figure 3(a). It is observed that when most of the parameters (>50%) are changed during updating, the accuracy gap between different pre-updating models is not significant (<0.2%). In contrast, when only a few parameters (10%~20%) are changed, the ratio of data for training pre-updating models has a more significant influence. For example, compared with the pre-updating model trained with 25% data, the pre-updating model trained with only 5% data leads to around 2.5% accuracy loss when 20% parameters are changed during updating, indicating that the performance of the pre-updating model has a direct influence to Tiny Updater. Specifically, when the pre-updating model is not trained with any data, it can be considered as a model with all parameters initialized with zero matrices, and thus Tiny Updater in this case degenerates to the common neural network pruning problem.

Multi-step Model Updating Previous experiments mainly show the result of one-step updating from using 25% to 100% training data. In this subsection, we show the performance of Tiny Updater in a multi-step updating, where the model is first trained with 5% data and then updated with 10%, 15%, 20%, 25%, and 100% training data, successively. As shown in Table 4, there is almost no accuracy loss when around 20% parameters are changed for each update. When there are around 10% parameters changed for each update, accuracy loss becomes significant with the increment on the update iterations (from -0.44% to -2.33%). This observation indicates that the accuracy loss tends to be accumulated in multi-step updating when an extremely low ratio of parameters is changed for updating.

Ablation Study Tiny Updater is mainly composed of two modules - pruning and knowledge distillation. Ablation studies on CIFAR10 and CIFAR100 are shown in Figure 4. It is observed that: (i) Tiny Updater achieves consistent effectiveness on all seven datasets. By changing around 20% parameters during updating, the neural networks trained with Tiny Updater achieve a similar performance with fully-updated models on all these datasets. (ii) On ImageNet, Tiny Updater achieves significant performance on the regular convolutional network such as ResNet [28], the efficient convolutional networks such as MobileNetv2 [71], and Swin Transformer [57]. (iii) With the proposed Tiny Updater, when a large ratio of parameters is changed, it even leads to higher performance than the fully-updated model. For instance, on Oxford Flowers datasets in Figure 3, the 84% updated model trained with Tiny Updater has 0.25% higher accuracy than the fully-updated model. We suggest this accuracy benefit is caused by the knowledge distillation loss in Tiny Updater. As pointed out by some previous research [2, 61], even if the student and the teacher have similar performance, knowledge distillation can still lead to consistent accuracy benefits.

Categorical Incremental Updating Figure 3(b) shows the effectiveness of Tiny Updater in the categorical incremental updating settings, where the pre-updating models are trained with 25% and 100% training data, respectively. Tiny Updater with random pruning indicates pruning the randomly selected channels.

Figure 4. Experiments on CIFAR with ResNet50. The pre-updating models and fully-updated models are trained with 25% and 100% training data, respectively. Tiny Updater with random pruning indicates pruning the randomly selected channels.

Figure 5. Experiments on CIFAR10 with the pre-updating models trained with different ratios of data.
Table 2. Experiments of multi-step updating on CIFAR10. Models are firstly trained with 5% training data and then updated by five times with 10%, 15%, 20%, 25%, and 100% training data by changing all, 10% and 20% parameters.

<table>
<thead>
<tr>
<th>Data Ratio</th>
<th>5%</th>
<th>10%</th>
<th>15%</th>
<th>20%</th>
<th>25%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-Updated</td>
<td>46.5</td>
<td>64.2</td>
<td>78.8</td>
<td>84.6</td>
<td>87.8</td>
<td>95.3</td>
</tr>
<tr>
<td>10% Params. Updated</td>
<td>46.5</td>
<td>63.8</td>
<td>78.3</td>
<td>83.2</td>
<td>85.9</td>
<td>92.9</td>
</tr>
<tr>
<td>20% Params. Updated</td>
<td>46.5</td>
<td>64.3</td>
<td>79.0</td>
<td>84.2</td>
<td>87.5</td>
<td>94.8</td>
</tr>
</tbody>
</table>

Tiny Updater on Pruned Models In this subsection, we study whether the proposed Tiny Updater can be utilized on a pruned model. Concretely, we first train a ResNet18 model with 25% training data on CIFAR10, which achieves 87.43% top-1 accuracy. Then, we apply the $L_1$-norm pruning technique to prune 53% neurons at the expense of 0.37% accuracy loss (87.43%→87.06%). Thirdly, the proposed Tiny Updater is applied to update this model with 100% training data, which achieves 94.22% at the expense of changing 24.3% parameters (87.06%→94.22%). In contrast, a pruned fully-updated model with 100% training data, which achieves 94.87%, is observed that when the same number of parameters are changed, Tiny Updater outperforms these two methods by a clear margin, indicating that Tiny Updater is more effective than directly applying previous methods.

Comparison with Finetuning and Adapter Finetuning and adapter-based methods are two well-known efficient training methods and the parameters changed in these two methods are also much fewer than global finetuning. Figure 4 gives the comparison between Tiny Updater and these two methods on CIFAR10, CIFAR100 with ResNet18. It is observed that when the same number of parameters are changed, Tiny Updater outperforms these two methods by a clear margin, indicating that Tiny Updater is more effective than directly applying previous methods.

4.2. Image-to-Image Translation with GANs

Experiment Setting We mainly evaluate Tiny Updater on image-to-image translation with CycleGAN for Horse→Zebra translation, and Pix2Pix for Edge→Shoe translation. CycleGAN is a typical unpaired image-to-image translation model, which has a ResNet-based generator trained with GAN loss and cycle consistency [94]. Pix2Pix is a typical paired image-to-image translation model, which has a U-Net architecture generator trained with conditional GAN loss and the $L_1$-norm loss [35]. Horse→Zebra is an unpaired dataset that translates natural images of horses to zebras and vice versa. It consists of 1,187 horse images and 1,474 zebra [93]. Edge→Shoe is a paired data that translates the edges of shoes to their corresponding natural images [34]. Fréchet Inception Distance (FID), which measures the distance between the feature distribution of the real and the generated images, is utilized as the metric for both datasets. A lower FID indicates the synthetic images have better quality.

Main Results Experiments of Tiny Updater on image-to-image translation are shown in Figure 7. It is observed that on the three image-to-image translation tasks, Tiny Updater is able to improve model performance by 5~10 FID when around 60% parameters are changed during updating. Besides, Tiny Updater can update the models by changing 40% parameters with no performance loss or by changing only 20% parameters with a slight FID increment. Qualitative analysis in Figure 6 shows that the pre-updating models can not transform the whole body of horses into the zebras, or remove all the stripes of zebras. In contrast, with Tiny Updater, the partially-updated model with only 20% parameters changed can address this problem and achieve comparable performance with the fully-updated models, indicating that Tiny Updater enables the neural network to be successfully updated by changing only a few parameters.

4.3. Video Recognition

Experiment Setting We evaluate Tiny Updater in video recognition datasets including UCF-101 [73] and Diving-48 [46] with video recognition models including SlowOnly [19] and Inception3D [9]. UCF-101 is an ac-
tion recognition dataset with 101 action classes over 13,000 video clips [73]. Diving-48 is a fine-grained video dataset of competitive diving. It has around 18,000 video clips belonging to 48 dive sequences [9]. Both top-1 and top-5 accuracy are reported.

**Main Results** Figure 8 shows the performance of Tiny Updater on video recognition. It is observed that on both SlowOnly and Inception3D (I3D), UCF-101 and Diving-48, the model with only 20% parameters changed trained by Tiny Updater achieves comparable and even higher performance than the fully-updated models. Moreover, the model with only 10% parameters changed during updating leads to only around 0.5% accuracy loss.

### 5. Discussion

#### 5.1. Choice of Pruning Granularity

Existing network pruning methods can be mainly divided into regular pruning (e.g. channel-wise pruning) and irregular pruning (e.g. element-wise pruning). Usually, irregular pruning can achieve a higher compression rate, while regular pruning is more friendly to hardware. In this paper, we prefer to use regular pruning in Tiny Updater and suggest that using irregular pruning may cause the problem of index overhead and memory access overhead.

**Index Overhead** When Tiny Updater is utilized for updating neural network-based software, users should download not only the parameters which are changed during updating but also an index file that records the corresponding position of each changed parameter in the neural network. Given a neural network with $N$ parameters, when irregular pruning is utilized, for each updated parameter, $\log_2 N$ bits storage space is required to record which layer, filter, and position in the kernel this parameter should be. Thus, an ignorable additional downloading overhead is caused. In contrast, the index overhead for channel-wise pruning is $\log_2 \frac{N}{C \times K^2}$ bits, which is small enough to be almost ignorable. Here $C$ and $K$ indicate the channel number and the kernel size, respectively.

**Memory Access Overhead** In Tiny Updater, irregular pruning can result in the updated parameters being distributed across various neural layers, channels, and filters, leading to their storage in different data blocks of the file system. Consequently, when updating the model with the downloaded parameters, almost all data blocks of the edge device need to undergo a writing operation, resulting in a significant memory access overhead. In contrast, regular pruning ensures that the updated weights are channel-wise and, therefore, can be stored in the same data blocks. Consequently, only these data blocks for the updated parameters require memory access, potentially reducing the updating overhead on edge devices.

### 6. Conclusion

This paper introduces the challenge of efficient model updating to reduce the download overhead during neural network-based software updating. To this end, we propose Tiny Updater, which aims to change only a small subset of the parameters during model retraining, thereby reducing the number of parameters that need to be downloaded by users. Tiny Updater comprises an iterative pruning technique that prunes the parameters of the fully-updated models to the pre-updating models and a knowledge distillation technique that treats the fully-updated models as teachers and the partially-updated models as students.

Extensive experiments were conducted on eleven datasets covering general image classification, fine-grained image classification, video recognition, paired image-to-image translation, and unpaired image-to-image translation. The results demonstrate that Tiny Updater can update neural network-based software by changing only 10% to 20% of the parameters with almost no loss in accuracy. In addition, the experimental results show the effectiveness of Tiny Updater in various scenarios, including categorical updating, multi-step updating, and pruning. We believe that this paper will encourage further research to address the challenge of efficient model updating on edge devices.
Table 3. Comparison among incremental learning, efficient finetuning, and the proposed efficient model updating.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Target</th>
<th>Pre / Post Tasks</th>
<th>Usage of Training Data</th>
<th>Constraints on Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental Learning</td>
<td>The human-like ability of incrementally learning new tasks without forgetting old tasks.</td>
<td>Different.</td>
<td>A small ratio of data is used for training pre-updating models. More data is used for updating. Data for pre-training data can not be accessed during updating (learning new tasks).</td>
<td>No direct constraints. Reducing the number of changed parameters sometimes lead to positive influence.</td>
</tr>
<tr>
<td>Efficient Finetuning</td>
<td>Finetuning pre-trained models for downstream tasks with low training overhead, and achieving better downstream performance.</td>
<td>Different.</td>
<td>Pre-updating models are pretrained on large-scale datasets, and then finetuned on much less data for downstream tasks. Usually do not access data for pre-training during finetuning.</td>
<td>No direct constraints. Do not finetune the whole model usually leads to no changes in pre-trained weights. Additional parameters in adapter layers and prompts are required.</td>
</tr>
<tr>
<td>Efficient Model Updating</td>
<td>Update models with new data by changing only a few parameters to reduce communication overhead.</td>
<td>Same.</td>
<td>A small ratio of data is used for pre-training models, and these data can still be accessed during updating.</td>
<td>Having direct constraints, the changed parameters should be as few as possible.</td>
</tr>
</tbody>
</table>

7. Acknowledgements
This research was partially supported by National Key R&D Program of China (2022YFB804103), Key Research and Development Program of Shaanxi (2021ZDLGY01-05), Tsinghua University Dushi Program, National Natural Science Foundation of China (20211710187), and Tsinghua University Talent Program.

A. Detailed Comparison
A detailed comparison among incremental learning, efficient finetuning and the proposed efficient model updating is shown in Table 3.

B. Detailed Experimental Setting
CIFAR10/100 are two datasets for low resolution general image classification with 50,000 images and 1,000 images in the training and validating set, respectively. ImageNet is a large-scale dataset for general image classification with images belonging to 1,000 categories. Stanford Car is a dataset for fine-grained image classification with car images belonging to 197 classes. Oxford Flower is a dataset for fine-grained image classification with 102 different categories of flowers common to the UK. Stanford Dog is a dataset for fine-grained image classification which consists of 102 different categories of flowers common to the UK. FGVC Aircraft is a dataset for fine-grained image classification which contains 10,000 images of aircrafts spanning 100 aircraft models.

C. Experiments on More Datasets and Tasks
Experimental results on more kinds of tasks and datasets are shown in Table 4 and Table 5. It is observed that our method leads to consistent good performance on cross-domain image classification, object detection and image super-resolution. Besides, our method outperforms previous efficient finetuning methods on ImageNet.
References


ings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016. 1, 6


[57] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In
Hossein Mobahi, Mehrdad Farajtabar, and Peter L. Bartlett.

Rashid Minhas, Abdul Adeel Mohammed, and Diego Ortego, Juan C. Sanmiguel, and José M. Martínez.


