# Learn-Prune-Share for Lifelong Learning

Zifeng Wang[1,*], Tong Jian[1,*], Kaushik Chowdhury[1], Yanzhi Wang[2], Jennifer Dy[1], Stratis Ioannidis[1]
*Department of Electrical and Computer Engineering*
*Northeastern University*
Boston, MA
[1]{zifengwang, jian, krc, jdy, ioannidis}@ece.neu.edu
[2]yanz.wang@northeastern.edu

***Abstract*—In lifelong learning, we wish to maintain and update a model (e.g., a neural network classifier) in the presence of new classification tasks that arrive sequentially. In this paper, we propose a *learn-prune-share* (LPS) algorithm which addresses the challenges of catastrophic forgetting, parsimony, and knowledge reuse simultaneously. LPS splits the network into task-specific partitions via an ADMM-based pruning strategy. This leads to *no forgetting*, while maintaining *parsimony*. Moreover, LPS integrates a novel selective knowledge sharing scheme into this ADMM optimization framework. This enables *adaptive knowledge sharing* in an end-to-end fashion. Comprehensive experimental results on two lifelong learning benchmark datasets and a challenging real world radio frequency fingerprinting dataset are provided to demonstrate the effectiveness of our approach. Our experiments show that LPS consistently outperforms multiple state-of-the-art competitors.**

***Index Terms*—Lifelong learning, Continual Learning, Model Pruning, Knowledge Reuse**

## I. Introduction

Human beings have a natural ability to adapt to different tasks sequentially without forgetting what they have learned. They can also seamlessly leverage knowledge learned from past tasks to tackle new tasks. This impressive ability is crucial for learning systems deployed in the real world. Lifelong learning [1] aims to develop models that mimic this human ability to learn continually without forgetting knowledge acquired earlier. In concrete terms, in a lifelong learning setting, we wish to maintain and update a model (e.g., a neural network classifier) in the presence of new classification tasks that arise sequentially. The model should both exhibit high accuracy on new tasks as well as perform well on old classification tasks, even if the old data is no longer accessible. However, learning algorithms are often designed to operate under stationary data distributions – typically, only a single task needs to be addressed. Under the lifelong learning setting, applying standard learning algorithms may lead to *forgetting* what has been learned on old tasks: this phenomenon, known as *catastrophic forgetting* [2], [3], results in severe performance degradation on old tasks after adapting to a new task.

A large body of work has been proposed to address catastrophic forgetting, using a varied arsenal of techniques [4]. Despite advances in lifelong learning, there are still limitations. Most of the methods, including, e.g., regularization-based [5]–[9] and rehearsal-based [10]–[13] methods, mitigate catastrophic forgetting under relatively restictive conditions, e.g., assuming a small number of highly related tasks. When tasks differ drastically, and the number of tasks grows, these methods suffer significant degradation. Another approach is to increase the model capacity (i.e., add parameters, neurons, layers, etc.), to accommodate new tasks, while preserving parts of the model for old tasks [14]–[16]. However, increasing complexity makes such methods prone to overfitting, and can be undesirable when models are to be deployed over memory-limited devices. Therefore, a competing objective of *parsimony* is desirable.

Another related challenge in lifelong learning is how to *reuse* learned knowledge to help the model learn future tasks better. Current research work often ignores this critical point by, e.g., independently considering different tasks [17], or by addressing it only partialy, e.g., using past parameters as an initialization during training [18]. However, the usefulness of knowledge gained from old tasks may depend on the relevance between old and new tasks. For example, a classifier trained for classifying dogs may be more helpful for classifying cats than digits. Thus, how to *adaptively select* useful past knowledge is critical for improving the performance on a new task.

Our proposed method, named *learn-prune-share* (LPS), is a novel deep learning framework aimed at addressing these challenges. LPS learns sequential tasks without experiencing catastrophic forgetting, by partitioning the neural network and dedicating portions to each task. It also prunes the neural network, thereby maintaining parsimony and avoiding overfitting. Finally, it selectively shares knowledge from old tasks and reuses them on new tasks. All of these happen simultaneously, in a unified optimization framework trained in an end-to-end fashion. Our contributions are as follows:

- We incorporate the state-of-the-art Alternating Direction Method of Multipliers (ADMM) based pruning strategy to solve the lifelong learning problem, maintaining a single parsimonious neural network model and eliminating catastrophic forgetting thoroughly.
- We design a novel knowledge sharing scheme, which learns to select useful knowledge from old tasks and adapt them to the current task. Our knowledge-sharing scheme is seamlessly integrated with our ADMM pruning strategy, and is trained jointly with the classifier parame-

[*] Z.Wang and T.Jian contributed equally to the paper.

ters. We make our code publicly available[1] to accelerate community contributions in this exciting topic.

- Our method, LPS, shows superior performance on two standard lifelong learning benchmark datasets as well as a challenging real world radio fingerprinting dataset. LPS beats state-of-the-art methods by a 2%–54% margin.

## II. Related Work

### A. Lifelong Learning

*Regularization-based* methods [5]–[9] limit plasticity of the network via regularization terms or by limiting the learning rate on parameters learned from previous tasks. While regularization-based methods mitigate catastrophic forgetting to some extent, performance on previous tasks gets increasingly worse when more diverse tasks are seen. By design, our method deals with catastrophic forgetting problem more effectively, as performance on previous tasks remains unchanged.

*Rehearsal-based* methods capture the data distribution in previous tasks by learning a generative model. When a new task arrives, data from previous tasks is simulated via the generative model and combined with current data to reinforce previous knowledge [10]–[13]. Though saving the generative model is less memory intensive than saving data, such models can still be big. Performance largely depends on the quality generative model on careful tuning of the mix of generated and new data. Our approach avoids the additional cost of training and storing an external generative model, again while experiencing no catastrophic forgetting.

*Expansion-based* methods accommodate new tasks by gradually increasing capacity of the model [14]–[16]. These methods generally outperform regularization and rehearsal based methods, which maintain a model with fixed capacity. However, the size of model parameters grows linearly with the number of tasks. This limits their practical usage, and makes them prone to overfitting. On the contrary, our approach fully exploits the potential of a fixed-capacity model.

Our method is closest to Continual Learning via Neural Pruning (CLNP) [19] and PackNet [18]. In these works, model pruning techniques are utilized to compress the original model iteratively to allocate free capacity for new tasks. However, both of these methods use simple threshold-based heuristics to prune the model with no structure constraint, resulting in a sparse, irregular matrix which limits further acceleration at inference time. Additionally, both of these methods consider tasks independently, ignoring the relationship between the current and previous tasks. In contrast, our approach adopts a systematic pruning strategy via Alternating Direction Method of Multipliers (ADMM), where structural constraints, e.g. filter pruning or column pruning [20], can be specified as needed. Moreover, our proposed novel knowledge inheritance scheme *adaptively select* weights shared from previous tasks to facilitate learning the current and future tasks. Our experimental results in Section V-B show that, due to these improvements, LPS outperforms these two algorithms.

### B. Neural Network Weight Pruning

The rich literature in neural network weight pruning can be categorized into *heuristic pruning algorithms* and *regularization-based pruning algorithms*. The former starts from the early work on irregular, unstructured weight pruning where arbitrary weights can be pruned. Han et al. [21] use an iterative algorithm to eliminate weights with small magnitude and perform retraining to regain accuracy. Guo et al. [22] incorporate connection splicing into the pruning process to dynamically recover the pruned connections that are found to be important. Later, heuristic pruning algorithms have been generalized to the more hardware-friendly structured sparsity schemes. A Transformable Architecture Search (TAS) [23] realizes the pruned network and knowledge is transferred from the unpruned network to the pruned version. Luo et al. [24] leverage a greedy algorithm to guide the pruning of the current layer with input information of the next layer, while Yu et al. [25] define a "neuron importance score" and propagate this score to conduct the weight pruning process.

Regularization-based pruning algorithms, on the other hand, have the unique advantage for dealing with structured pruning problems through group Lasso regularization [26]. Early work [27], [28] incorporate $\ell_1$ or $\ell_2$ regularization in loss function to solve filter/channel pruning problems. Zhuang et al. [29] introduce an $\ell_2$-norm variant indicating the number of selected channels in each layer. A number of subsequent works are dedicated to making the regularization penalty a dynamic and "soft" term. The method in [30] selects filters based on $\ell_2$-norm and updates the filters that have been previously pruned, while [31], [32] incorporate the advanced optimization solution framework Alternating Direction Methods of Multipliers (ADMM) to achieve dynamic regularization penalty, thereby improving accuracy. We take advantage of the state-of-the-art ADMM-based pruning strategy by [31] and [32]. Moreover, we integrate a novel selective knowledge sharing scheme into the ADMM optimization framework, captured by learnable masks. Furthermore, our whole pipeline can be trained in an end-to-end fashion performing learn, prune, share simultaneously through ADMM.

## III. Problem Formulation

In supervised lifelong learning, we are given a sequence of datasets $\mathcal{D} = \{\mathcal{D}^1, \mathcal{D}^2, \ldots, \mathcal{D}^n\}$, where each dataset $\mathcal{D}^t = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m_t}$, $t = 1, \ldots, n$, contains tuples of the input feature $\mathbf{x} \in \mathbb{R}^d$ and its corresponding label $y \in \mathbb{N}$. Each dataset corresponds to a distinct classification task: labels $y \in \mathbb{N}$ are disjoint across datasets $\mathcal{D}^t$. Datasets are revealed *sequentially*: dataset $\mathcal{D}^t$ becomes accessible only at the $t$-th task, which corresponds to, e.g., moving to a new environment. Our goal is to train a classifier sequentially on the datasets such that it achieves good performance on all tasks.

Formally, we are given a feature extractor $f_W : \mathbb{R}^d \to \mathbb{R}^{d'}$ parameterized by $W \in \mathbb{R}^m$. After the network is trained on $\mathcal{D}^t$, along with a task-specific output layer, its parameters $W$ are updated. If $W^t$ are the parameters of the feature extractor at task $t$, a final classifier is obtained after training the extractor

[1] `https://github.com/neu-spiral/LPSforLifelong`

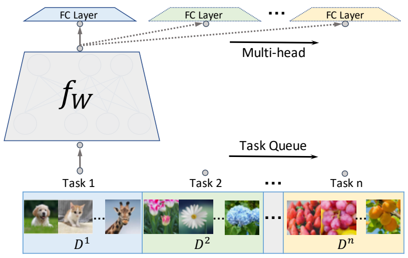

Fig. 1: An illustration of supervised lifelong learning. A feature map $f_W$ is trained sequentially on datasets $\mathcal{D} = \{\mathcal{D}^1, \mathcal{D}^2, \ldots, \mathcal{D}^n\}$, where each dataset becomes accessible only at the corresponding task. A fully connected layer at the end of the classifier, denoted as one 'head', is attached to $f_W$ to handle the new task. This is commonly referred to as a "multi-head" output later: faced with sequential $n$ tasks, the classifier branches in $n$ heads/output layers.

(and the $n$ correponding output layers) on all datasets in $\mathcal{D}$ sequentially, as illustrated in Fig. 1. The overall performance of $f_{W^n}$ is then assessed via the average classification accuracy on separate testsets, one for each task $t$. Note that, at test time, we are aware of which task/environment $f_{W^n}$ is operating over, so that we can classify using the appropriate output layer.

While the problem setting is straightforward, we need to point out three desiderata that must be addressed by a supervised lifelong learning solution.

**Catastrophic Forgetting.** Catastrophic forgetting is the widely reported phenomenon [2], [3] that models, especially neural networks, tend to "forget" information from previous tasks when incorporating knowledge from new tasks. This is observed in accuracy performance degradation on previous tasks after being exposed to new tasks. Addressing catastrophic forgetting is a central issue, and the main objective of most lifelong learning algorithms [14]–[16], [18], [19].

**Parsimony.** Due to limited computation and memory in real world applications, but also to avoid overfitting, the model $f_W$ should be as compact as possible. It is therefore desirable to maintain a single model and adapt it to various tasks, instead of, e.g., training multiple specialized models.

**Knowledge Reuse.** Related to both parsimony and catastrophic forgetting, beyond memorizing knowledge acquired from previous tasks, we also want to exploit it when encountering new tasks. For example, parts of the model could be shared across tasks; this leverages relevant/reusable features across tasks, leading to further parsimony and avoiding overfitting, while also ameliorating catastrophic forgetting. Thus, it is important to strike a balance between reuse vs. growth or plasticity in a network, in a way that performance improves.

## IV. Learn-Prune-Share

We propose a learn-prune-share (LPS) algorithm, a novel deep learning framework for lifelong learning incorporating neural network pruning via ADMM. Our method maintains a single neural network for the sequence of tasks, while **learning** the tasks, **pruning** the neural network, and **sharing** knowledge among tasks; these three happen synergistically. Departing from conventional regularization-based or network-expansion-based methods, LPS fully exploits the capacity of the neural network by splitting it into disjoint partitions specialized for each task via pruning; in turn, this mitigates catastrophic forgetting. Simultaneously, to exploit earlier knowledge obtained from previous tasks, LPS shared parameters between different partitions of the network, in an adaptive, tunable fashion.

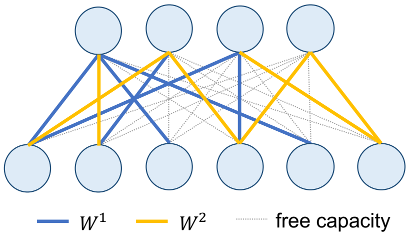

Fig. 2: Split of network weights at task 2. Task designated weights $W^1$, $W^2$ have disjoint support, and a lot of excess capacity in the network remains free.

### A. Architecture Overview

We assume that we are given a legacy neural network architecture $f_W : \mathbb{R}^d \to \mathbb{R}^{d'}$ (e.g., ResNet [33]), parameterized by weights $W \in \mathbb{R}^m$. Recall that the support of a vector is the set of its non-zero coordinates. Our solution satisfies the following two properties: first, at the conclusion of task $t$, the weights of the network are partitioned into task-specific weights $W^1, W^2, \ldots, W^t \in \mathbb{R}^m$ that have disjoint supports. Formally, for all $1 \leq i, j \leq t$ with $i \neq j$:

$$\mathrm{supp}(W^i) \cap \mathrm{supp}(W^j) = \emptyset. \tag{1}$$

Second, these disjoint weights do not exhaust the entire representation capacity of the network: the union of their supports is smaller than $m$. The remaining weights are treated as excess capacity, to be utilized in future tasks. Formally, let

$$\bar{W}^t = \textstyle\sum_{i=1}^t W^i \in \mathbb{R}^m, \tag{2}$$

be the sum of the task-specific weights.[2] Then,

$$\mathrm{supp}(\bar{W}^t) = \textstyle\bigcup_{i=1}^t \mathrm{supp}(W^i) \leq m. \tag{3}$$

Figure 2 illustrates the weight split for a single layer at task $t = 2$. Weights $\bar{W}^2 = W_1 + W_2$ are partitioned to two classes $W^1$ and $W^2$ with disjoint support. Moreover, the excess capacity (the complement of $\bar{W}^2$'s support) can be used for future tasks.

Under this configuration, to make predictions for task $t$, our network uses $W^t$, i.e. the portion of the network representing task-specific knowledge, as well as *as many of the weights $\bar{W}^{t-1}$ dedicated to previous tasks as we wish to leverage*. Formally, the network we use for task $t$ has weights

$$W^t + M^t \odot \bar{W}^{t-1}, \quad \text{for } t = 1, \ldots, n, \tag{4}$$

where $\odot$ represents element-wise multiplication and $M^t \in \{0,1\}^m$ are a set of learnable *knowledge sharing masks*.

[2] As $W^i$, $i = 1, \ldots, t$ have disjoint supports, $\bar{W}^t$ can also be thought of as their superposition.

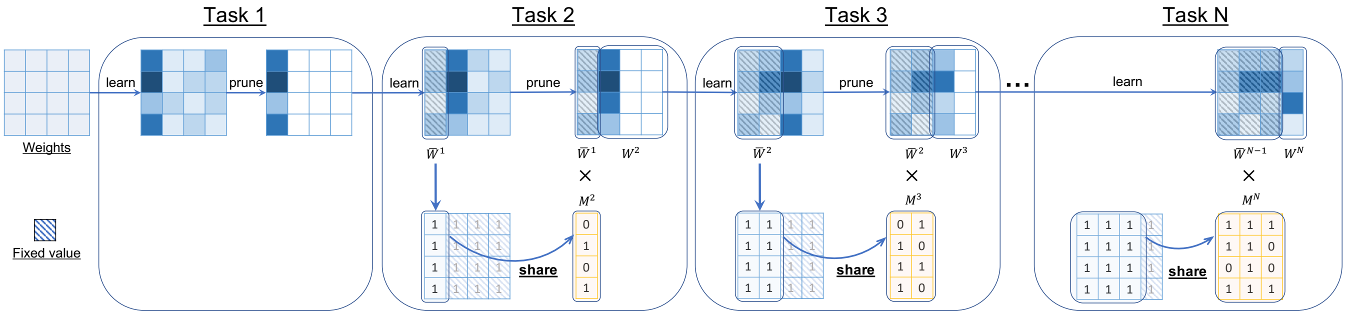

Fig. 3: Overview of the proposed LPS method. For each task $t$, given $\bar{W}^{t-1}$ from previous tasks till $(t-1)$, we *learn* the task, *prune* the neural network to obtain task specific weights $W^t$, and *share* knowledge among tasks via mask $M^t$, simultaneously. Note that for task 1, we only need to learn $W^1$, as there is no previous knowledge yet; and for the last task N, we do not need to prune unless there is requirement of leaving free capacity for future tasks.

Our solution, and in particular the weight design in Eq. (4), has several advantages, each addressing directly the issues of catastrophic forgetting, parsimony, and knowledge reuse. First, our approach *does not experience any catastrophic forgetting*. This is precisely because additional tasks are accommodated in excess capacity; classification for earlier tasks (also through Eq. (4)) remains unaltered. Second, by utilizing only a portion of the overall capacity of the network, we attain parsimony. As we discuss below, this happens at almost no accuracy loss: we learn the small-support parameters $W^i$, $i = 1, \dots, t$ through state-of-the art pruning methods. Finally, the use of masks $M^t \in \{0,1\}^m$ enables arbitrary levels of reuse: setting them to 1 fully reuses weights learned from previous tasks, while setting them to 0 limits the network for task $t$ to only its dedicated weights. Note that this flexibility comes at the expense of parsimony, as we also need to keep track of masks for each task. As these are binary, however, they are not as memory-intensive as the model weights.

### B. The Learn-Prune-Share (LPS) Algorithm

Our learn-prune-share algorithm learns task-specific weights $W^t$ as well as knowledge-sharing masks $M^t$ as the datasets $\mathcal{D}_t$ are revealed. It is an iterative process, summarized in Figure 3. At each task, we use the full excess capacity of the network to train a dense network. Using a state-of-the-art pruning method, we reduce this to weights with small support $W^t$; simultaneously, we determine how much of the old weights to reuse via mask $M^t$. This process is repeated until we run out of tasks.

Formally, at each task $t$, the input to the algorithm consists of (a) earlier weights from previous tasks 1 through $(t-1)$, i.e., $\bar{W}^{t-1} = \sum_{i=1}^{t-1} W^i \in \mathbb{R}^m$, as well as, (b) the dataset of task $t$ , i.e., $\mathcal{D}^t$. Our goal is to learn *sparse, small-support* task-specific weights $W^t$, as well as the knowledge-sharing mask $M^t$. Note that for task 1, we only need to learn $W^1$, as there is no previous knowledge yet. As our pruning happens layer-wise, we introduce the following notation. We re-write the weights and masks as $W = \{W_l\}_{l=1}^L \in \mathbb{R}^m$ and $M = \{M_l\}_{l=1}^L \in \{0,1\}^m$ where $W_l, M_l$ are the weights and masks, respectively, corresponding to the $l$-th layer, for $l = 1, \dots, L$. We denote the loss of a network with weights $W$ under dataset $D$ as $\mathcal{L}(W, W_{L+1}; \mathcal{D})$, where $W_{L+1} \in \mathbb{R}^{P_{L+1} \times Q_{L+1}}$ is the final (classification) layer. In light of Eq. (4), we formulate the learning process determining $W^t, W^t_{L+1}, M^t$ at task $t$ as an optimization problem:

$$\begin{aligned}
\underset{W, W_{L+1}, M}{\text{Min.}:} \quad & \mathcal{L}\left(W + M \odot \bar{W}^{t-1}, W_{L+1}; \mathcal{D}^t\right), && \text{(5a)}\\
\text{subj. to:} \quad & W_l \in S_l^t, \quad l = 1, \cdots, L, && \text{(5b)}\\
& M_l \in S_l'^t, \quad l = 1, \cdots, L, && \text{(5c)}\\
& \text{supp}(W) \cap \text{supp}(\bar{W}^{t-1}) = \emptyset, && \text{(5d)}\\
& \text{supp}(M) \subseteq \text{supp}(\bar{W}^{t-1}), && \text{(5e)}\\
& W \in \mathbb{R}^m, W^t_{L+1} \in \mathbb{R}^{P_{L+1} \times Q_{L+1}}, && \text{(5f)}\\
& M \in \{0,1\}^m && \text{(5g)}
\end{aligned}$$

where $S_l^t$ are *sparsity constraints* on $W_l^t$, and $S_l'^t$ are *knowledge-sharing constraints on* $M_l^t$. We describe both in detail below, in Sections IV-C and IV-D, respectively.

The constraint in Eq. (5d) enforces that weights are indeed disjoint: the weights of $W^t \in \mathbb{R}^m$ are taken from the current excess capacity pool – the complement of $\text{supp}(W^{t-1})$. Similarly, the constraint in Eq. (5e) enforces that the knowledge-sharing mask $M \in \{0,1\}^m$ are applied to the past weights $W^{t-1}$ only. Note that, together, they imply that $W^t$ and $M^t$ have disjoint supports. Finally, the fully connected classifier/output weights $W^{L+1}$ are unconstrained.

### C. Task-Specific Weight Constraints

To obtain $W^t$, we need to create constraints on $W = \{W_l\}_{l=1}^L \in \mathbb{R}^m$ in Prob. (5) that enforce sparsity. Recall that we denote the weights of the $l$-th layer of our neural network as $W_l$. In convolutional layers, the weight for $l$-th layer is represented by a four-dimensional tensor, where dimensions $p_l, q_l, r_l, s_l \in \mathbb{N}$ correspond to the number of filters, number of channels, filter width, and filter height, respectively. In fully connected layers, weights are $P_l \times Q_l$ matrices, where $P_l$ and $Q_l$ represent the input and output layer size, respectively. We nevertheless assume that all layers are represented in a GEneral Matrix Multiplication operations (GEMMs) format, which is a standard practice in tensor framework implementations: that

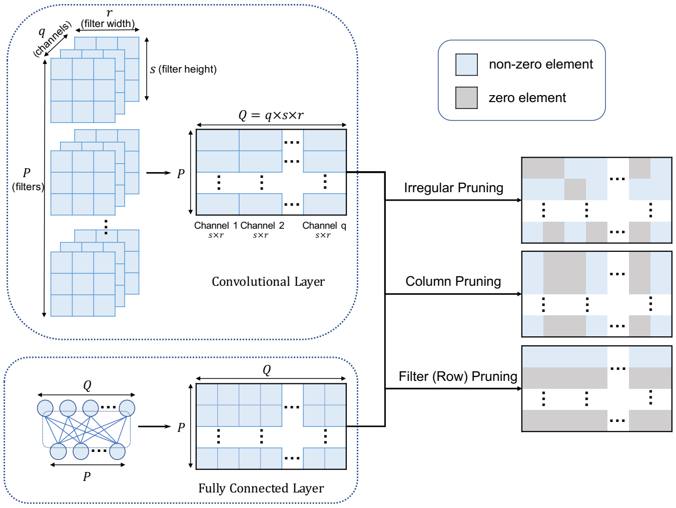

Fig. 4: Pruning strategy illustration. By converting weights to the format of GEneral Matrix Multiplication operations (GEMMs), we represent both CV and FC layers via the (reshaped) weight matrix $W \in \mathbb{R}^{P\times Q}$. We can then choose from ***irregular*** or ***structured*** (i.e. column and filter) pruning.

is, we assume all tensors are reshaped to two dimensional $P_l \times Q_l$ matrices. This is already the case for fully connected layers; for convolutional layers, the reshaping can take the form $P_l = p_l$ and $Q_l = q_r \cdot r_l \cdot s_l$. We thus assume every layer is represented by a (reshaped) weight matrix $W_l \in \mathbb{R}^{P_l\times Q_l}$, as illustrated in Figure 4. Note that, under this assumption, the total number of weights in the model is $m = \sum_{l=1}^{L} P_l \cdot Q_l$.

Under this representation, we consider the following sets of constraints $S_l^t$ for layer $l$:

**Irregular Pruning.** For irregular pruning, we have:

$$S_l^t = \{W_l \in \mathbb{R}^{P_l\times Q_l} \mid \|W_l\|_0 \leq \alpha_l^t\}, \qquad (6)$$

where $\|\cdot\|_0$ the size of $W_l$'s support (i.e., the number of non-zero elements), and $\alpha_l^t \in \mathbb{N}$ is a constant limiting the proportion of non-zero elements. Intuitively, this implies that $W_l$ has no more than $\alpha_l^t$ non-zero elements.

**Structured Pruning.** Given $\phi$ a Boolean predicate, let $\mathbb{1}_\phi$ to be 1 if $\phi$ is true, and 0 otherwise. Moreover, given matrix $W_l \in \mathbb{R}^{P_l\times Q_l}$, let $[W_l]_{:,q} \in \mathbb{R}^{P_l}$ be the $q$-th column of $W_l$. In ***column pruning***, the constraint set $S_l^t$ is defined as:

$$S_l^t = \{W_l \in \mathbb{R}^{P_l\times Q_l} \mid \left(\sum_{q=1}^{Q_l} \mathbb{1}_{([W_l]_{:,q}\neq \mathbf{0})}\right) \leq \alpha_l^t\}, \qquad (7)$$

where $\alpha_l^t \in \mathbb{N}$. This enforces that the number of non-zero *columns* in the $l$-th layer's GEMM representation does not exceed $\alpha_l^t$. A similar constraint can be placed on filters/rows of $W_l$ to form structured ***filter pruning***, which enforces that the number of non-zero *filters* does not exceed $\alpha_l^t$.

All three types of constraints (irregular, column, and filter pruning) are illustrated in Fig. 4. They all lead to disjoint supports if used consistently across tasks: for example, filter pruning ends up partitioning rows of the GEMM representation of every later, column pruning partitions columns, etc., while irregular pruning partitions individual matrix entries.

### D. Knowledge-Sharing Mask Constraints

To control knowledge sharing, we impose a sparsity constraint on $M$ as well, allowing only $\beta_l^t \in \mathbb{N}$ of entries in the mask to be non-zero. Formally:

$$S_l^{'t} = \{M_l \in \{0,1\}^{P_l\times Q_l} \mid \|M_l^t\|_0 = \beta_l^t\}. \qquad (8)$$

Adjusting the "sharing parameter" $\beta_l^t$ allows us to limit the proportion of old weights shared (i.e., the non-zero elements of $M_l$). By forcing $M_l$ to be sparse, we force training to select the most beneficial weights for the current task from previously learned weights. Sharing parameter $\beta_l^t$ also conveys the *usefulness* of previous knowledge: e.g. when tasks are similar, previous knowledge would indeed be useful for subsequent tasks, thus $\beta_l^t$ should be big; conversely, for dissimilar tasks we expect fewer sharing opportunities.

### E. Solving LPS via ADMM

The optimization problem defined in Eq. (5) for LPS has non-convex constraints, and solving it via standard stochastic gradient descent is not possible. We use the widely deployed Alternating Direction Method of Multipliers (ADMM) [34], that has been extensively applied in pruning literature [31], [35]. For completeness, we describe the ADMM solution to Problem (5) in detail in Appendix A. In short, ADMM decomposes the original non-convex problem with constraints into subproblems that can be solved separately. It alternates between (a) standard gradient descent with a quadratic proximal penalty (Eq. (13)), that forces the solution to be close to a value in the (non-convex) constraint space, and (b) an orthogonal projection operation to the constraint space (Eq. (14a)). Hence starting from full weights $W$ and masks $M$ set to 1, we can progressively prune and constrain both, producing a feasible solution at convergence. Most importantly, the weights and masks can be trained jointly and dynamically.

From an implementation standpoint, to incorporate our constraints to ADMM, it suffices to produce polynomial-time functions that compute the orthogonal projection into constraints (5b) – (5c). For (5b), polynomial algorithms are well known for irregular, column, and filter pruning constraints [31]. For example, for irregular pruning, the orthogonal projection of a matrix $Z \in \mathbb{R}^{P_l\times Q_l}$ to set $S_l^t$ given by Eq. (6) can be computed by keeping the $\alpha_l^t$ entries of $Z$ of largest absolute value intact, and setting the rest to zero. For column pruning (Eq. (7)), projection of $Z$ to can $S_l^t$ be computed by similarly keeping the $\alpha_l^t$ columns with largest $\ell_2$ norm intact, and setting all other rows to $\mathbf{0}$.

Our mask constraint (8) is more complex, as projection requires not only enforcing sparsity exactly, but also that the values of the matrix become binary. Nevertheless, we can compute the projection of $Z \in \mathbb{R}^{P_l\times Q_l}$ to $S_l^{'t}$ in polynomial time via the following steps:

Sort elements of matrix $Z$ from smallest to largest;
Map the largest $\beta_l^t$ entries to 1; set the rest entries to 0.

We prove the correctness of this algorithm in Appendix B.

## V. Experiments

In our experiments, (a) we show that our method outperforms current state-of-the-art methods on both benchmark and real datasets; (b) we assess the importance of the knowledge-sharing mask under different task settings; and (c) we explore how different pruning strategies affect the prediction accuracy.

### A. Experimental Setting.

**Datasets.** To evaluate the performance of our approach empirically, we experiment with two standard lifelong learning benchmark datasets, permuted MNIST [36], [37] and split CIFAR-10/100 [38], and a real world radiofrequency fingerprinting dataset (split RF) [39], summarized in Table I. The original MNIST dataset [36], [37] contains 28 × 28 black and white images of handwritten digits of 10 classes. Following [6], we construct 10 tasks by applying the same random permutation across all MNIST images, using a different permutation for each task. CIFAR-10 [38] comprises 10 classes of 32x32 colour images. CIFAR-100 is just like CIFAR-10 in image format and total number of images, but has 100 classes. Following [6], we set the first task as the whole CIFAR-10 dataset. We then create 5 additional tasks, each containing 10 consecutive classes from the CIFAR-100 dataset. Finally, the split RF dataset [39], [40] contains radio transmissions from 50 WiFi devices recorded in the wild. We randomly partition these 50 classes into 5 tasks.

**Lifelong Learning Methods.** We compare LPS to the following methods:

*Elastic Weight Consolidation (EWC) [5]:* EWC applies Laplace Approximation to estimate the importance scores of parameters for previous tasks and uses a quadratic regularizer weighted by the importance scores.

*Intelligent Synapses (IS) [6]:* IS uses an importance score based regularizer similar to EWC. However, a path integral based method is proposed to evaluate the importance score.

*Learning without Forgetting (LwF) [7]:* LwF maintains responses for previous tasks via a knowledge distillation loss.

*Deep Generative Replay (DGR) [10]:* DGR uses generative adversarial networks (GAN) [41] to mimic the data distribution for each task. A generator is updated at every task to incorporate its data distribution. A corresponding classifier is trained using the mixture of generated and new data.

*Gradient Episodic Memory (GEM) [11]:* GEM proposes an episodic memory saving a portion of previous data and use the loss on this data a constraint when training a new task.

*PackNet [18]:* PackNet iteratively prunes the model to accommodate new tasks by removing parameters of smaller magnitude heuristically. Similar formulation is proposed by [17] under a lifelong learning setting.

We use the implementation from the original authors for all methods, including the recommended hyperparameter settings or tuning strategies. The same network architectures are used among all methods for fair comparison.

**Architectures.** We implement different architectures for permuted MNIST, split CIFAR-10/100, and split RF, respectively. The architecture for permuted MNIST dataset [6] contains two hidden layers, each with 2000 neurons and ReLU activations. For split CIFAR-10/100 dataset, we use the default CIFAR-10 architecture from Keras [6]. For split RF dataset, we use ResNet50-1D [42], which is the 1D-convolutional version of ResNet50, targeting inputs as 2D fixed-length sequences. For all three architectures, we learn the biases and batch normalization parameters for the first task and keep these terms fixed for subsequent tasks.

TABLE I: Dataset and Parameter Summary.

| Stat. & Param. | Datasets | | | |
|---|---|---|---|---|
| | Permuted MNIST | Split CIFAR | | Split RF |
| | | $n = 1$ | $n \neq 1$ | |
| # tasks ($n$) | 10 | 6 | | 5 |
| # classes per task | 10 | 10 | | 10 |
| # train samples per task | 60,000 | 50,000 | 5,000 | 1,410 |
| # test samples per task | 10,000 | 10,000 | 1,000 | 550 |
| $\alpha_l^t$ (% total layer params) | 10% | 50% | 10% | 20% |
| $\beta_l^t$ (% total $\bar{W}_l^{t-1}$ params) | 90% | 92% | | 90% |
| Pruning strategy | Irregular | Irregular | | Column |
| LPS Epochs (warm-up/ADMM/final) | 30/90/30 | 200/600/200 | | 20/60/20 |
| Architecture | Two FC layers | CIFAR-10 | | ResNet50-1D |
| # params ($m$) | 5,568,000 | 884,576 | | 15,901,568 |
| # layers ($L$) | 2 | 5 | | 49 |

**LPS Implementation.** For each task, we run LPS in three phases. In the warm-up phase, we first train a $W$ over the full free parameters with $M = 1$. In the ADMM phase, we then prune the network Eq. (11). In the final stage, we do a final projection to the constraint sets of both masks and weights, and retrain the weights, changing only non-zero values. We set all $\rho_i = 10^{-3}$ and increase by a factor of 10 at equal intervals during ADMM iterations. We use the following hyperparameters, which we determine using a validation set. Unless otherwise noted, sparsity parameters $\alpha_l^t$ and $\beta_l^t$ are as shown in Table I. We explored the impact of both in Section V-B. For all experiments, we use a batch size of 128 and Adam [43] as an optimizer with default values and initialize the learning rate to 0.001. Our proposed LPS approach is implemented in Python using PyTorch [44] and NVIDIA CUDA support. All experiments are carried out on an Tesla V100 GPU with 32 GB memory and 5120 cores.

**Evaluation Metrics.** We evaluate the final obtained model (associated with masks and multi-head output layers) on all tasks testsets via (Top-1) accuracy.

### B. Results on Benchmark Datasets

**Effectiveness of the proposed LPS approach.** Table II shows the overall performance, in terms of the final average accuracy across all tasks, of all lifelong learning methods. For reference purposes, we also include the accuracy attained when training a full-capacity (non-parsimonious) single model separately for each task (SM). LPS outperforms all competitors across all datasets. Most methods perform well on permuted MNIST; the margin is wider on the remaining two datasets, that are more challenging. To further scrutinize the performance of LPS across tasks, we show in Table III-IV the per task accuracy.

TABLE II: Overall performance on three benchmark datasets. For all the methods, we report the final average accuracy (%) across all tasks. We include SM (column 2) for reference purpose, which trains a full-capacity single model separately for each task. LPS parameters are set as in Table I.

| Datasets | Methods | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | SM | EWC | IS | LwF | DGR | GEM | PackNet | LPS |
| Permuted MNIST | 98.80 | 96.81 | 97.52 | 68.22 | 90.73 | 93.03 | 98.14 | **98.58** |
| Split CIFAR-10/100 | 75.14 | 71.13 | 74.97 | 54.68 | 63.61 | 66.05 | 77.79 | **80.13** |
| Split RF | 81.15 | 37.01 | 42.63 | 27.75 | 48.27 | 68.38 | 79.37 | **81.22** |

TABLE III: Split CIFAR-10/100: For all the methods, we report the task-specific, and the final average accuracy (%) across all tasks. LPS parameters are set as in Table II.

| Methods | Tasks | | | | | | |
|---|---|---|---|---|---|---|---|
| | task 1 | task 2 | task 3 | task 4 | task5 | task 6 | Avg. |
| SM | 82.32 | 75.40 | 70.20 | 75.90 | 71.70 | 75.30 | 75.14 |
| EWC | 71.23 | 72.50 | 69.25 | 71.34 | 67.52 | 74.93 | 71.13 |
| IS | 74.59 | 74.28 | 74.19 | 75.54 | 75.58 | 75.62 | 74.97 |
| LwF | 40.32 | 56.77 | 48.60 | 53.94 | 60.04 | 68.43 | 54.68 |
| DGR | 64.36 | 62.01 | 63.02 | 67.34 | 65.28 | 59.64 | 63.61 |
| GEM | 68.52 | 65.34 | 63.88 | 70.12 | 65.23 | 63.23 | 66.05 |
| PackNet | 82.33 | 79.30 | 73.90 | 78.80 | 74.30 | 78.10 | 77.79 |
| LPS | **82.97** | **80.00** | **76.50** | **79.90** | **78.40** | **83.00** | **80.13** |

TABLE IV: Split RF: For all the methods, we report the task-specific, and the final average accuracy (%) across all tasks. LPS parameters are set as in Table II.

| Methods | Tasks | | | | | |
|---|---|---|---|---|---|---|
| | task 1 | task 2 | task 3 | task 4 | task 5 | Avg. |
| SM | 76.33 | 73.50 | 85.30 | 85.60 | 85.00 | 81.15 |
| EWC | 25.73 | 35.32 | 30.85 | 45.81 | 47.24 | 37.01 |
| IS | 27.08 | 40.72 | 37.25 | 50.66 | 57.34 | 42.63 |
| LwF | 14.62 | 20.37 | 23.45 | 33.58 | 46.72 | 27.75 |
| DGR | 43.50 | 49.37 | 43.87 | 50.25 | 54.38 | 48.27 |
| GEM | 67.24 | 63.45 | 68.53 | 70.26 | 72.44 | 68.38 |
| PackNet | 78.15 | 74.14 | 82.56 | 80.54 | 81.45 | 79.37 |
| LPS | **78.33** | **77.55** | **84.19** | **82.63** | **83.39** | **81.22** |

Interestingly LPS outperforms all competitors across all tasks on both datasets; we also observed this on the 10 tasks of the permuted MNIST, which we omit for brevity. Overall, *our LPS approach achieves both the best average and the best task-specific accuracy for all three datasets*.

We further observe that regularization-based methods like EWC and IS perform relatively well on benchmarks, while they fail on split RF. One possible explanation may be that when tasks are more diverse and model is large, regularizers do not suffice to keep the learned information. Evidence of forgetting is present in LwF, for split CIFAR, and almost all methods (except LPS and PackNet) on split RF. This is expected, as both LPS and PackNet are immune to forgetting.

We also observe that LSP even outperforms the full-capacity SM trained from scratch on each task for split CIFAR-10/100 and split RF, and is very close to it over permuted MNIST. This happens despite the fact that it uses only a small fraction of the $m$ parameters used by SM, indicating that it avoids overfitting. Also, we see a clear benefit of reuse of parameters across tasks in split CIFAR (Table III): by partially utilizing past weights, prediction on later tasks improves under LPS compared to SM.

**Share Parameter Effects.** We further explore the impact of knowledge-sharing in Figure 5. The figure shows how average and per task accuracy changes as we modify $\beta_l^t$: the $x$-axis is the share ratio, indicating the ratio of the parameter over the total number of past weights per layer on the CIFAR dataset. The optimal value is at 92%. Moreover, we clearly see that a large reduction in sharing has a bigger impact on later tasks-which otherwise would benefit from knowledge reuse.

We also show the results of models with no (0%) and full (100%) share on all datasets as well as our best performing model with selective sharing in Table V. We follow the same parameter searching strategy as in split CIFAR-10/100 to get the best performing model on validation set. Interestingly, for all three datasets, we observe the best performance achieved by setting share ratio around 90%. This also indicates that many (not all) past weights are valuable or meaningful for new tasks.

To explore this notion of knowledge re-use further, we conducted an experiment in which tasks vary drastically. To do so, we construct a 5-task "mixed" dataset, where tasks 1,3,5 are from the MNIST dataset, with different permutation patterns and tasks 2, 4 both contain 10 different classes from CIFAR-100. Images from permuted MNIST are augmented to RGB images by repeating 3 channels using the original image and resized to $32 \times 32$ to be compatible with CIFAR images. Similar to Figure 5, Figure 6 shows the effect of the sharing ratio on the mixture dataset. Not surprisingly, the behavior is quite different from Fig. 5. The highest accuracy (89.22%) is achieved by 20% share, which demonstrates that LPS does adaptively select useful knowledge for the current task. Note that, faced with these dissimilar tasks, full share (88.15%) performs even worse than no share (88.23%), indicating the share strategy choice should be flexible and guided by the inter-task similarity.

**Comparing different pruning strategies.** We compared three different pruning strategies (i.e., column, filter, and irregular pruning) on split CIFAR-10/100 and split RF datasets, summarized in Table VI and Table VII, respectively. Both irregular and column pruning obtain satisfactory performance, achieving 80.13% and 79.56% on split CIFAR-10/100, 80.55% and 81.22% on split RF, respectively. However, filter pruning reflects an unstable performance, obtaining 68.11% and 80.12% on split CIFAR-10/100 and split RF datasets, respectively.

**Impact of Model Capacity.** Figure 7 measures how model capacity usage affects the accuracy on the split CIFAR-10/100 dataset. For this experiment, instead of using the whole model capacity for the 6 tasks, we use only a fraction (e.g., $x\%$) of the full model by the $n$-th task, leaving $1-x\%$ parameters free for future growth; all other parameters are set as in Table I. Figure 7 shows the impact on average and per task accuracy as we vary fraction $x$. We clearly observe that a model performs better when more capacity is available. Nevertheless, accuracy performance is also robust to this shrinkage – it achieves 75.32% accuracy with only 50% model capacity, which is even

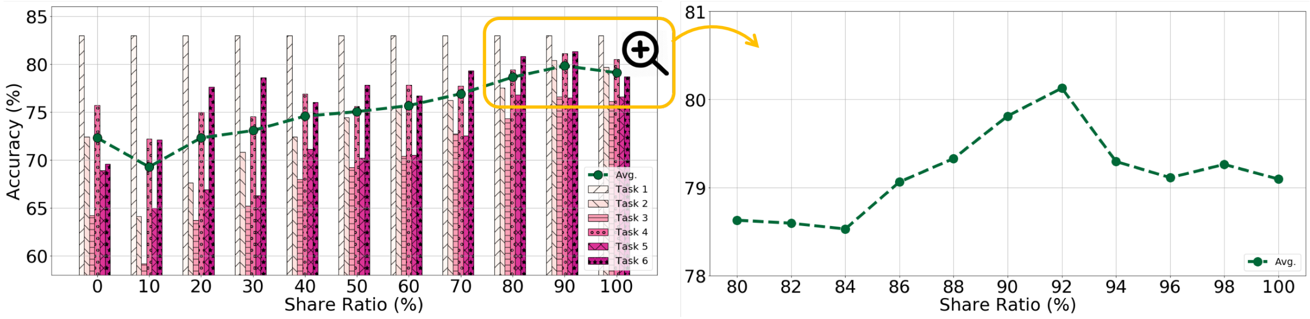

Fig. 5: Split CIFAR-10/100: An exploration of how average and per task accuracy changes as we modify the share ratio $\beta_l^t$. The x-axis is the share ratio, indicating the ratio of the parameter over the total number of past weights per layer. For each share ratio, we represent the task specific and average accuracy as the colored bar and the green dot, respectively. The optimal is at 92% share, depicted right.

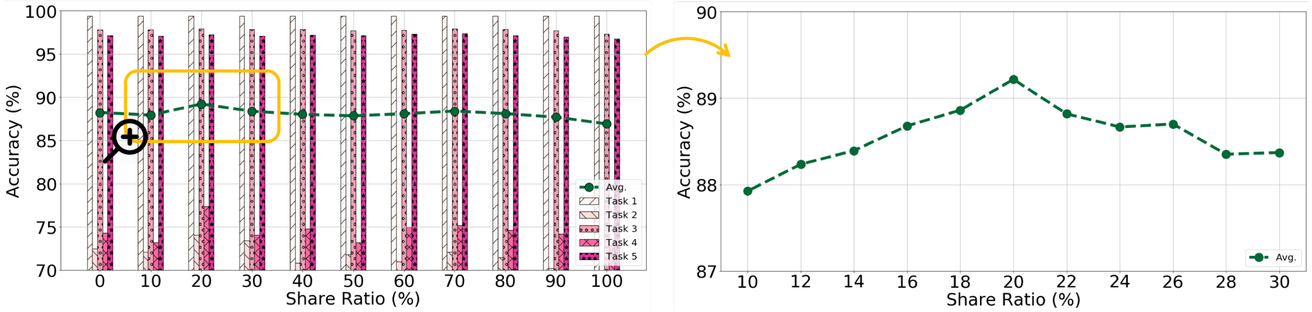

Fig. 6: Mixed Dataset: An exploration of how average and per task accuracy changes as we modify the share ratio $\beta_l^t$ on non-similar tasks. The optimal value is at 20% share, depicted in detail in the right figure. Less knowledge reuse performs even better, demonstrating LPS does adaptively select useful knowledge for the current task, and indicating the share strategy choice should be guided by the inter-task similarity.

TABLE V: LPS with no (0%), full (100%) and selective share on three benchmark datasets. For selective share, we follow the same parameter searching strategy as in split CIFAR-10/100 to get the best performing model. To make a fair comparison, we start experiments from the learned model on task 1 (no previous knowledge yet), then sequentially train this model on remaining tasks with different share ratio $\beta_l^t$.

| Datasets | $\beta_l^t$ | Tasks | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | task 1 | task 2 | task 3 | task 4 | task5 | task 6 | task 7 | task 8 | task 9 | task 10 | Avg. |
| Permuted MNIST | 0% | **98.92** | **98.77** | 98.47 | 98.51 | **98.58** | 98.49 | 98.29 | 97.91 | 97.78 | 85.82 | 97.15 |
| | 100% | **98.92** | 98.56 | 98.51 | 98.39 | 98.35 | 98.24 | 98.26 | 98.19 | 98.25 | 98.14 | 98.38 |
| | 90% | **98.92** | 98.68 | **98.71** | **98.64** | 98.55 | **98.61** | **98.49** | **98.51** | **98.42** | **98.23** | **98.58** |
| Split CIFAR-10/100 | 0% | **82.97** | 72.40 | 64.20 | 75.70 | 68.90 | 69.60 | - | - | - | - | 72.30 |
| | 100% | **82.97** | 79.70 | 76.10 | **80.50** | 76.60 | 78.70 | - | - | - | - | 79.10 |
| | 92% | **82.97** | **80.00** | **76.50** | 79.90 | **78.40** | **83.00** | - | - | - | - | **80.13** |
| Split RF | 0% | **78.33** | 77.33 | 83.29 | 81.90 | 82.20 | - | - | - | - | - | 80.61 |
| | 100% | **78.33** | **77.59** | **84.93** | 81.90 | 83.12 | - | - | - | - | - | 81.17 |
| | 90% | **78.33** | 77.55 | 84.19 | **82.63** | **83.39** | - | - | - | - | - | **81.22** |

better than the best non-pruning method IS (74.97%) with full model capacity. Surprisingly, at only 10% of the total capacity of the network, accuracy does not collapse, but still remains above 72.5%. This indicates that our method has the potential capacity to scale to even more future tasks.

## VI. Conclusions and Future Work

In this paper, we propose the learn-prune-share (LSP) algorithm for lifelong learning. Our method maintains a parsimonious neural network model and achieves exact no forgetting by splitting the network into task-specific partitions via ADMM-based pruning method. Moreover, a novel selective knowledge sharing scheme is integrated seamlessly into the ADMM optimization framework to address knowledge reuse. Experiments on permuted MNIST, split CIFAR10/100 and split RF demonstrates our approach achieves significant improvement over the state-of-the-art methods. Future directions include applying more advanced pruning strategies on the lifelong learning problem and exploring how to measure the capacity of a model quantitatively.

## VII. Acknowledgements

The authors gratefully acknowledge support by the National Science Foundation (grant CCF-1937500).

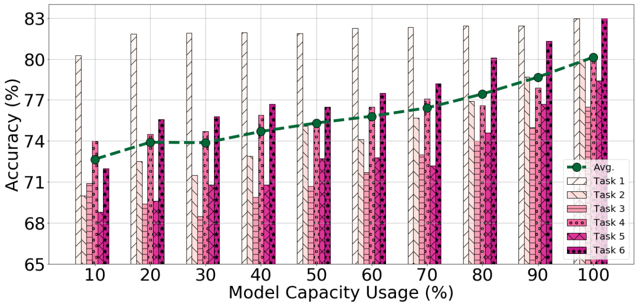

Fig. 7: Split CIFAR-10/100: To demonstrate our LPS method has the potential capacity to scale to more future tasks, we use only a certain fraction (e.g., $x\%$) of the full model by the 6-th task, leaving $1 - x\%$ parameters free for future growth. The x-axis is the fraction of the model capacity usage. As it can be observed, LPS achieves 75.32% average accuracy with only 50% model capacity, which is even better than the best non-pruning method IS (74.97%) with full model capacity.

TABLE VI: Three pruning strategies on split CIFAR-10/100.

| Prun. Appr. | $\beta_l^t$ | Tasks | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | task 1 | task 2 | task 3 | task 4 | task5 | task 6 | Avg. |
| SM | | 82.32 | 75.40 | 70.20 | 75.90 | 71.70 | 75.30 | 75.14 |
| Irregular | 0% | **82.97** | 72.40 | 64.20 | 75.70 | 68.90 | 69.60 | 72.30 |
| | 100% | **82.97** | 79.70 | 76.10 | **80.50** | 76.60 | 78.70 | 79.10 |
| | 92% | **82.97** | **80.00** | **76.50** | 79.90 | **78.40** | **83.00** | **80.13** |
| Column | 0% | **82.04** | 68.80 | 56.50 | 71.00 | 63.90 | 63.00 | 67.54 |
| | 100% | **82.04** | 80.80 | 76.20 | 80.30 | 76.40 | 77.90 | 78.94 |
| | 92% | **82.04** | **80.90** | **76.30** | **80.60** | **77.10** | **80.40** | **79.56** |
| Filter | 0% | **79.95** | 56.50 | 50.40 | 62.20 | 54.60 | 55.80 | 59.91 |
| | 100% | **79.95** | 60.20 | 60.00 | 60.40 | 58.90 | 61.10 | 63.43 |
| | 92% | **79.95** | **62.10** | **61.70** | **67.70** | **66.50** | **70.70** | **68.11** |

TABLE VII: Three pruning strategies on split RF.

| Prun. Appr. | $\beta_l^t$ | Tasks | | | | | |
|---|---|---|---|---|---|---|---|
| | | task 1 | task 2 | task 3 | task 4 | task 5 | Avg. |
| SM | | 76.33 | 73.50 | 85.30 | 85.60 | 85.00 | 81.15 |
| Irregular | 0% | **78.33** | **75.14** | 83.74 | 82.19 | 73.03 | 78.49 |
| | 100% | **78.33** | **75.14** | 84.01 | 79.71 | 82.20 | 79.88 |
| | 90% | **78.33** | 74.21 | **84.56** | **83.00** | **82.65** | **80.55** |
| Column | 0% | **78.33** | 77.33 | 83.29 | 81.90 | 82.20 | 80.61 |
| | 100% | **78.33** | **77.59** | **84.93** | 81.90 | 83.12 | 81.17 |
| | 90% | **78.33** | 77.55 | 84.19 | **82.63** | **83.39** | **81.22** |
| Filter | 0% | **77.59** | 70.32 | 82.64 | 80.36 | 82.39 | 78.66 |
| | 100% | **77.59** | 73.65 | 82.90 | 80.44 | 82.85 | 79.49 |
| | 90% | **77.59** | **74.54** | **83.64** | **81.72** | **83.12** | **80.12** |

## References

[1] S. Thrun and T. M. Mitchell, "Lifelong robot learning," *Robotics and autonomous systems*, vol. 15, no. 1-2, pp. 25–46, 1995.

[2] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*. Elsevier, 1989, vol. 24, pp. 109–165.

[3] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions." *Psychological review*, vol. 97, no. 2, p. 285, 1990.

[4] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, 2019.

[5] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[6] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *ICML*, 2017, pp. 3987–3995.

[7] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.

[8] C. V. Nguyen, Y. Li, T. D. Bui, and R. E. Turner, "Variational continual learning," in *ICLR*, 2018.

[9] R. Aljundi, F. Babiloni, M. Elhoseiny, M. Rohrbach, and T. Tuytelaars, "Memory aware synapses: Learning what (not) to forget," in *ECCV*, 2018, pp. 139–154.

[10] H. Shin, J. K. Lee, J. Kim, and J. Kim, "Continual learning with deep generative replay," in *NeurIPS*, 2017, pp. 2990–2999.

[11] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continual learning," in *NeurIPS*, 2017, pp. 6467–6476.

[12] G. M. van de Ven and A. S. Tolias, "Generative replay with feedback connections as a general strategy for continual learning," in *COSYNE Workshop*, 2019.

[13] O. Ostapenko, M. Puscas, T. Klein, P. Jähnichen, and M. Nabi, "Learning to remember: A synaptic plasticity driven framework for continual learning," in *CVPR*, 2019.

[14] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[15] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," in *ICLR*, 2018.

[16] T. J. Draelos, N. E. Miner, C. C. Lamb, J. A. Cox, C. M. Vineyard, K. D. Carlson, W. M. Severa, C. D. James, and J. B. Aimone, "Neurogenesis deep learning: Extending deep networks to accommodate new classes," in *IJCNN*, 2017, pp. 526–533.

[17] S. Golkar, M. Kagan, and K. Cho, "Continual learning via neural pruning," *arXiv preprint arXiv:1903.04476*, 2019.

[18] A. Mallya and S. Lazebnik, "Packnet: Adding multiple tasks to a single network by iterative pruning," in *CVPR*, 2018, pp. 7765–7773.

[19] S. Golkar, M. Kagan, and K. Cho, "Continual learning via neural pruning," *arXiv preprint arXiv:1903.04476*, 2019.

[20] S. Ye *et al.*, "Progressive weight pruning of deep neural networks using admm," *arXiv preprint arXiv:1810.07378*, 2018.

[21] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[22] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *NeurIPS*, 2016, pp. 1379–1387.

[23] X. Dong and Y. Yang, "Network pruning via transformable architecture search," in *NeurIPS*, 2019, pp. 759–770.

[24] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *ICCV*, 2017, pp. 5058–5066.

[25] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in *CVPR*, 2018, pp. 9194–9203.

[26] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *ICLR*, 2019.

[27] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *NeurIPS*, 2016, pp. 2074–2082.

[28] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017, pp. 1389–1397.

[29] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *NeurIPS*, 2018, pp. 875–886.

[30] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *IJCAI*, 2018.

[31] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *ECCV*, 2018, pp. 184–199.

[32] T. Li, B. Wu, Y. Yang, Y. Fan, Y. Zhang, and W. Liu, "Compressing convolutional neural networks via factorized convolutional filters," in *CVPR*, 2019, pp. 3977–3986.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[34] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.

[35] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *ASPLOS*, 2019, pp. 925–938.

[36] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits, 1998," *URL http://yann. lecun. com/exdb/mnist*, vol. 10, p. 34, 1998.

[37] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013.

[38] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[39] T. Jian, B. C. Rendon, E. Ojuba, N. Soltani, Z. Wang, K. Sankhe, A. Gritsenko, J. Dy, K. Chowdhury, and S. Ioannidis, "Deep learning for rf fingerprinting: A massive experimental study," in *IEEE Internet of Things Magazine*, 2020.

[40] A. Gritsenko, Z. Wang, T. Jian, J. Dy, K. Chowdhury, and S. Ioannidis, "Finding a 'new'needle in the haystack: Unseen radio detection in large populations using deep learning," in *DySPAN*. IEEE, 2019, pp. 1–10.

[41] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *NeurIPS*, 2014, pp. 2672–2680.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[44] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

## Appendix A
## Solving Problem (5) via ADMM

To begin with, constraints (5d), (5c) are easy to satisfy: we basically partition variables of $W$ and $M$ to sets $\mathrm{supp}(\bar{W}^{t-1})$ and its complement, and only optimize over the appropriate set (the complement of $\mathrm{supp}(\bar{W}^{t-1})$ for $W$ and $\mathrm{supp}(\bar{W}^{t-1})$ for $M$). We thus ignore these constraints below. We similarly omit $W_{L+1}$, which is unconstrained and can be learned via SGD. Rewriting the loss as $\mathcal{L}(W, M)$, we convert the non-convex optimization problem formulated in (5) into the ADMM form by introducing auxiliary variables $Z_l$ and $Y_l$ for constraints (5b) and (5c) respectively:

$$\begin{aligned} \min_{W,M}: \quad & \mathcal{L}(W,M) + \textstyle\sum_{l=1}^{L} g_l(Z_l) + \sum_{l=1}^{L} h_l(Y_l), && (9a) \\ \text{subject to:} \quad & W_l = Z_l, \quad l = 1, \cdots, L, && (9b) \\ & M_l = Y_l, \quad l = 1, \cdots, L, && (9c) \end{aligned}$$

where $g_l(\cdot)$ and $h_l(\cdot)$ correspond to the indicator functions for constraints (5b) and (5c) respectively, i.e.,:

$$g_l(Z_l) = \begin{cases} 0, \text{ if } Z_l \in S_l^t, \\ +\infty, \text{ o.w.}, \end{cases} \quad h_l(Y_l) = \begin{cases} 0, \text{ if } Y_l \in S_l^{\prime t}, \\ +\infty, \text{ o.w.} \end{cases} \tag{10}$$

The augmented Lagrangian of (9) is:

$$\begin{aligned} &\mathcal{L}_a(W, M, Z, U, Y, K) = \mathcal{L}(W, M) \\ &+\textstyle\sum_{l=1}^{L}\left\{g_l(Z_l)+\rho_l \operatorname{Tr}(U_l^\top(W_l-Z_l))+\frac{\rho_l}{2}\|W_l-Z_l\|_F^2\right\} \quad (11) \\ &+\textstyle\sum_{l=1}^{L}\left\{h_l(Y_l)+\tau_l \operatorname{Tr}(K_l^\top(M_l-Y_l))+\frac{\tau_l}{2}\|M_l-Y_l\|_F^2\right\} \end{aligned}$$

where $\rho_l$ and $\tau_l$ are penalty terms, and $U_l \in \mathbb{R}^{P_l \times Q_l}$ and $K_l \in \mathbb{R}^{P_l \times Q_l}$ are dual variables, rescaled by $\rho_l$ and $\tau_l$, respectively. ADMM proceeds iteratively as follows; at the $n$-th iteration:

$$\begin{aligned} (W, M)^{n+1} &= \underset{W,M}{\arg\min}\, \mathcal{L}_a(W,M,Z^n,U^n,Y^n, K^n) && (12a) \\ (Z, Y)^{n+1} &= \underset{Z,Y}{\arg\min}\, \mathcal{L}_a(W^{n+1},M^{n+1},Z, U^n,Y,K^n) && (12b) \\ U^{n+1} &= U^n + W^{n+1} - Z^{n+1} && (12c) \\ K^{n+1} &= K^n + M^{n+1} - Y^{n+1}. && (12d) \end{aligned}$$

The problem (12a) is equivalent to:

$$\begin{aligned} \min_{\boldsymbol{W},\boldsymbol{b}} \quad & \mathcal{L}(W, M) + \textstyle\sum_{l=1}^{L} \frac{\rho_l}{2}\|W_l - Z_l^n + U_l^n\|_F^2 \\ & + \textstyle\sum_{l=1}^{L} \frac{\rho_l}{2}\|M_l - Y_l^n + K_l^n\|_F^2. \end{aligned} \tag{13}$$

The first term in (13) is a standard DNN loss while the second and the third terms are quadratic and differentiable. Thus, this subproblem can be solved by classic stochastic gradient descent. Problem (12b) is equivalent to:

$$\begin{aligned} Z_l^{n+1} &= \Pi_{S_l^t}\left(W_l^{n+1} + U_l^n\right), \quad \text{for all } l = 1, \ldots, L, && (14a) \\ Y_l^{n+1} &= \Pi_{S_l^{\prime t}}\left(M_l^{n+1} + K_l^n\right), \quad \text{for all } l = 1, \ldots, L, && (14b) \end{aligned}$$

where $\Pi_{S_l^t}, \Pi_{S_l^{\prime t}}$ are the Euclidean projections onto sets $S_l, S_l^{'}$, respectively.

## Appendix B
## Proof the correctness of Mask Projector

For simplicity, we prove this for the projection to the set: $S = \{x \in \{0,1\}^n : \|x\|_0 = k\}$. i.e., the set of $n$ binary elements containing k zeros. Let $y \in \mathbb{R}^n$, then $\Pi_S(y)$ is computed by: (a) sort all elements $y_i \in y$ from smallest to largest; (b) set the $k$ largest values to 1 an the rest to 0. We make use of the following lemma.

**Lemma 1.** *For $a, b \in \mathbb{R}$, where $a \leqslant b$, $a^2 + (1-b)^2 \leqslant (a-1)^2 + b^2$.*

This can be easily proved by considering all positional cases of $a, b \in \mathbb{R}$. Let $\hat{y} \in S$ be the solution of the algorithm, and $y^* \in \arg\min_{x \in S} \|x - y\|_2$ be an optimal solution. Assume indices are order based on the elements of $y$, as in the algorithm. Let $i$ be the first position at which $\hat{y}_i \neq y_i^*$. Then, $y_i$ is mapped to 0 in $\hat{y}$ and $y_i$ is mapped to 1 in $y_i^*$. Moreover, as both have exactly $k$ ones, there must be a $j$ such that (i) $y_j \geqslant y_i$, (ii) $\hat{y}_j = 1$, and (iii) $y_j^* = 0$. By the lemma, since $y_j \geqslant y_i$, we have $y_i^2 + (y_j - 1)^2 \leqslant (y_i - 1)^2 + y_j^2$. So, setting $y_i^* = 0$ and $y_j^* = 1$ would only improve distance from $y$. As $y^*$ is optimal, this swap must maintain optimality; repeating this procedure as long as there exist indices at which $\hat{y}$ and $y^*$ differ will convert $y^*$ to $\hat{y}$, while maintaining optimality. □