Neural Weight Search for Scalable Task Incremental Learning

Jian Jiang and Oya Celiktutan

Department of Engineering, King's College London, London, UK
{jian.jiang, oya.celiktutan}@kcl.ac.uk

Abstract

Task incremental learning aims to enable a system to maintain its performance on previously learned tasks while learning new tasks, solving the problem of catastrophic forgetting. One promising approach is to build an individual network or sub-network for future tasks. However, this leads to an ever-growing memory due to saving extra weights for new tasks and how to address this issue has remained an open problem in task incremental learning. In this paper, we introduce a novel Neural Weight Search technique that designs a fixed search space where the optimal combinations of frozen weights can be searched to build new models for novel tasks in an end-to-end manner, resulting in a scalable and controllable memory growth. Extensive experiments on two benchmarks, i.e., Split-CIFAR-100 and CUB-to-Sketches, show our method achieves state-of-the-art performance with respect to both average inference accuracy and total memory cost. [1]

1. Introduction

The last decade has demonstrated the power of deep learning approaches, achieving superior performance in many machine vision tasks. However, modern machine learning algorithms for robots assume that all the data is available during the training phase. On the other hand, the real world is highly varied, dynamic, and unpredictable. It is infeasible to collect enough amount of data to represent all the aspects of the real world. Therefore, robots must learn from their interactions with the real world continuously. Motivated by this, incremental learning (also known as lifelong learning and continual learning) is an emerging research area, which aims to design systems that can gradually extend their acquired knowledge over time through learning from the infinite stream of data [7,10,14,16,26,36].

Incremental learning remains a challenging open problem because it demands minimal performance loss for old tasks and a minimal increase in model storage. In other words, the models should be able to adapt to novel tasks efficiently and effectively, while not significantly underperforming on the previously learned tasks, which is known as the problem of catastrophic forgetting.

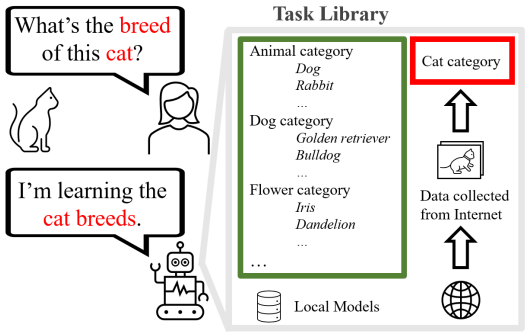

Figure 1: Envisioned one practical application of the task incremental learning in real-world human-robot interaction settings. To enable such practical applications, this paper aims to address the problem of memory growth.

There is a significant body of work on incremental learning [1, 23]. These methods can be divided into two widely used categories based on the learning scenario, namely, class-incremental learning (CIL) and task-incremental learning (TIL). Generally, CIL methods build on a single network model that needs to sequentially learn a series of tasks. One advantage is that they do not require the task id during inference, unlike TIL methods. However, the single shared model will inevitably cause degradation in performance for previously learned tasks. Moreover, state-of-the-art CIL methods, namely, replay-based methods [16, 17], require extra memory for saving exemplars per task, e.g., typically a task has 5 classes and 20 images are saved per class, which will lead to a significant increase in memory as the number of tasks becomes larger.

TIL methods, on the other hand, learn a separate model for each new task, inherently addressing the problem of for-

[1] Implementation: https://github.com/JianJiangKCL/NeuralWeightSearch

getting. One downside is that task id is needed during inference. A solution to this is illustrated in Figure 1, where a robot can interactively query its user to determine the task id in both training and inference, which brings about practical applications in human-robot interaction [21]. However, building an individual network or sub-network for new tasks leads to an ever-growing memory due to saving extra weights and how to address this issue lies at the heart of TIL research, especially, considering the limitations of robotics platforms, i.e., space constraints.

Due to its relevance to human-robot interaction, in this paper, we focus on task incremental learning setting, with the goal of alleviating the memory growth problem as a robot encounters new tasks sequentially. To this end, a line of research has focused on dynamically expanding the network for task incremental learning by creating a new model or a subnetwork to learn new information while allowing old models to maintain their performance. Among these methods, Progressive Neural Network (PNN) [29] freezes the previously learned models and continuously adds new models for new tasks. One downside of PNN is that it results in substantial growth of parameters with the increasing number of tasks. Recent works [19, 20, 35] have focused on learning task-specific kernel masks to transfer and adapt the backbone model to a certain task. PiggyBack [19] aims to learn binary element-wise kernel masks for new tasks. Though, there are two main limitations to this approach. First, binary values might limit the representation capability. Second, the fixed backbone hampers the learning of new tasks. PackNet [20] aims to solve this problem by freeing up some of the existing weights, which are unused by previous tasks, for learning new tasks. However, the backbone model of PackNet can also run out of learnable weights and might become a 'fixed' backbone eventually. To address this issue, Compacting, Picking, and Growing (CPG) [8], grows and prunes the backbone of PiggyBack for incremental learning, which however has a tideous iterative training process and may lead to a significant increase in the storage required for saving models. Taken together, scalable task incremental learning has remained an open problem, particularly, how to utilise previously learned knowledge for new tasks and grow the models in a controllable manner.

This work approaches this open problem by tapping into a scalable technique, called Neural Weight Search (NWS), which significantly alleviates the memory growth problem while achieving state-of-the-art performance. Differently from the existing methods [19, 20, 35], where a backbone model is used, NWS discards the backbone but maintains frozen layer-wise pools of grouped weights. For each new task, it builds a new model by searching the pools for the optimal combinations of grouped weights. This search is efficiently conducted with the help of a set of temporary kernel weights that are not used for processing inputs and are discarded after the training. Figure 2 illustrates the workflow of our proposed approach.

In summary, our main contributions are as follows: (i) We propose a new problem setting named *Neural Weight Search* (NWS). Analogously to Neural Architecture Search (NAS), NWS *automatically searches pretrained weights to build a network*. (ii) The backbone-free design is novel in task incremental learning. Unlike the conventional kernel mask-based methods, where the backbone model is fixed or partially fixed, our method discards the backbone but maintains layer-wise pools of kernel weights, allowing more representation capacity. (iii) NWS enables the re-utilization of weights when building models and achieves scalable performance. Compared to the state-of-the-art method [35], our memory gain can reach up to 82% (including the memory required for saving the pools) and achieves better average accuracy by a margin of 1.9% on two existing benchmarks.

2. Related Work

There is a significant body of work on incremental learning, which can be divided into three broad categories, namely, regularization-based (e.g., [11, 36]), architecture-based (e.g., [8, 19, 20, 24, 25, 35]), and replay-based (e.g., [3, 9, 10, 15, 17, 18, 27]). Our proposed approach is at the intersection of two categories, namely, *regularization-based*, and *architecture-based* methods. As mentioned before, there are two widely used settings in incremental learning, namely, class-incremental learning (CIL) [3, 9, 10, 15, 17, 18, 27], task-incremental learning (TIL) [8, 19, 20, 35]. In this work, we focus on the task incremental learning setting.

TIL approaches are generally architecture-based methods (also known as parameter-isolation methods). These methods learn an individual model or a partially original model (e.g., shared low-level layers and individual high-level layers) for each task. Overall, these methods suffer from an uncontrollable growth of the memory when new models are saved. For example, Rusu *et al*. proposed Progressive Neural Network (PNN) [29] to continuously expand the network by generating a new network for each task while fixing the previously learned networks, which resulted in an uncontrolled growth in parameters and hence poor scalability.

Recent methods like Piggyback [19], PackNet [20], CPG [8], and KSM [35] have aimed to alleviate this problem by introducing learnable masks with a single backbone model. The weights of a new model are generated by multiplying the masks with corresponding weights in the backbone model. PiggyBack [19] fixes the backbone network and learns binary element-wise masks for kernels. First, real-valued masks are generated, which have the size same as kernels. Then a predefined threshold is applied to obtain binary masks. Such masks, namely 'hard masks', also result in poor scalability as kernels that can be learned for new

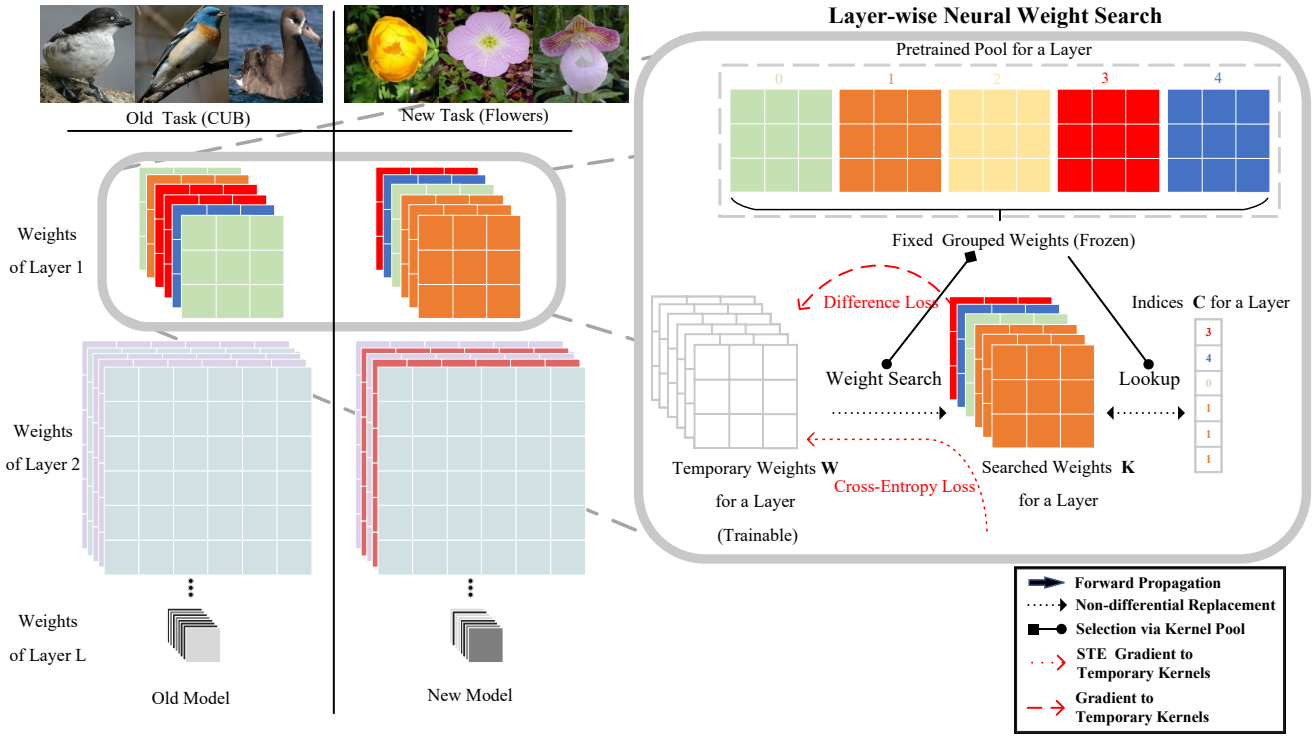

Figure 2: The illustration of the Neural Weight Search algorithm. In both training and inference, only searched kernel weights are used to process inputs. The temporary kernel weights can be regarded as a scaffold for building a neural network model, which can be discarded after training.

tasks are constrained by the fixed backbone model. PackNet [20], building upon PiggyBack, uses a strategy to prune weights that are not used by old tasks, allowing to free up those parameters for learning future tasks. However, PackNet does not add more kernels and it reaches a bottleneck when no more parameters are left to be released. The ability of incremental learning is the same as PiggyBack, when it runs out of learnable weights. Hence, PackNet advances PiggyBack but it is still not scalable in the long term.

Inspired by the aforementioned methods, CPG [8] adopts the structure of Piggyback but it enables adaption of the network by iteratively introducing more kernels for new coming tasks and pruning the learned model. The iterative expanding and pruning operations continue until the model reaches a pre-defined inference performance for a task. For example, the pre-defined performance can be defined as the inference accuracy of a set of baseline models individually fine-tuned for the corresponding task. However, CPG has two main limitations. First, prior knowledge about baseline performance is not usually available beforehand. Second, it has a tedious and demanding iterative training process. A recent method called KSM [35] uses the setting the same as Piggyback but utilises soft kernel-wise masks that combine binary and real values instead of using binary element-wise kernel masks. Compared to Piggyback, soft kernel-wise masks enhance the incremental learning ability, allowing kernels to adapt to different tasks using richer representations. KSM reaches the state-of-the-art performance on the Split-CIFAR-100 [13] benchmark and CUB-to-Sketches benchmark [2,5,12,22,30,34]. However, KSM also relies on a fixed backbone model that limits the learning of representations. Unlike the aforementioned kernel-mask-based methods, where the backbone model is fixed or partially fixed, our method discards the use of the backbone. We maintain frozen pretrained kernel weights that are saved in the form of layer-wise pools and these weights can be efficiently reused in various combinations and orderings for different tasks, maximising the plasticity and significantly reducing memory growth.

3. Problem Definition

3.1. Task Incremental Learning (TIL)

In a general TIL setting, there are T incremental phases where t_{th} phase introduces a new v_t-way classification task with training data $\mathbf{x}_t$ and labels $\mathbf{y}_t$, where $y_t \in \{0, 1, 2, ..., v_t - 1\}$. For each task, we aim to learn an individual model f_t parameterized by learnanble weights $\boldsymbol{\theta}_t$ with the objective, $\arg\min_{\boldsymbol{\theta}_t} \mathcal{L}(f_t(\mathbf{x}_t), \mathbf{y}_t)$, where $\mathcal{L}$ is a classification loss. During the inference, each task-specific model f_t is evaluated on the corresponding test data set $\mathbf{x}_t^{test}$ and $\mathbf{y}_t^{test}$.

3.2. Neural Weight Search

To address the limitations of existing approaches (see Section 2), we define a new problem setting named Neural Weight Search (NWS) for TIL. Instead of relying on a fixed or partially fixed backbone and learning a mask for each new task, NWS aims to build a new model by searching for an optimal combination of weights from fixed and stored weights pretrained on a large-scale image dataset. These weights can be reused (without updating the values of weights) for any upcoming new task. Treating each weight scalar as an element to search for would lead to significantly large search space. In practice, we search for grouped weights and a group of weights can either be a convolution kernel, a filter or even a layer. Briefly, we first design a fixed search space of indexed grouped neural weights. Then, we search for an optimal combination of grouped weights in the search space to build a new model for a new task, where the same grouped weights can be shared within the same task as well as across different tasks.

As illustrated in Figure 2, for each layer l, we hold a search space of n^l indexed groups of weights $\mathbf{M}^l = \{\mathbf{m}_1, \mathbf{m}_2, ..., \mathbf{m}_{n^l}\}$, which is called a "layer-wise pool" or simply a "pool" in our formulation. Let's consider that we need to learn d^l groups of weights for this layer, denoted by $\mathbf{W}^l = \{\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_{d^l}\}$. NWS builds this layer by searching for an optimal combination ($\boldsymbol{comb}$) of weights $\mathbf{K}^l = \{\mathbf{k}_1, \mathbf{k}_2, ..., \mathbf{k}_{d^l}\}$ where $\mathbf{k}_i \in \mathbf{M}^l$. Note that the search processes of different layers are conducted simultaneously with the objective $\arg\min_{\boldsymbol{comb}} \mathcal{L}(f_{\mathbf{K}}(\mathbf{x}), \mathbf{y})$. Because new models are formed with indexed grouped weights, if there are a number of models, NWS can greatly reduce memory cost by saving their combination indices as well as the search space. In this paper, we group weights in the form of convolution kernels, e.g., for a convolution kernel with size 3×3, 9 weight values are grouped together. We have demonstrated the effectiveness of our approach with varying convolutional network architectures.

To the best of our knowledge, neural weight search as introduced in this paper has not been explored before. NWS is completely different from both 'kernel search' in mixed-integer linear programming and 'kernel search optimization' used in kernel methods like SVM. NWS is analogous to Neural Architecture Search (NAS). NAS designs a search space of model components beforehand and obtains an optimal model architecture by evaluating the combinations of these components. NWS designs a search space for neural weights and *automatically searches for an optimal combination of weights to build a network*.

4. Proposed Method

Our proposed method has two main components, namely, Neural Weight Search (NWS) algorithm (Section 4.1), and design of the search space (Section 4.2).

4.1. NWS Algorithm

As mentioned in Section 3.2, we divide weights into groups such that each group represents a convolution kernel. We define a convolution kernel as a $k \times k$ matrix in the float domain. To build a convolution layer l, we search a layer-wise pool for an optimal combination of kernels, $\mathbf{K}^l \in \mathbb{R}^{d^l \times k \times k}$, where $d^l = d^l_{in} * d^l_{out}$ and d^l_{in}, d^l_{out} are the number of input channels and output channels respectively. A layer-wise pool $\mathbf{M}^l \in \mathbb{R}^{n^l \times k \times k}$ is learned and fixed during the pretraining stage, where n^l is the number of kernels in the pool (see Section 4.2). The kernels in the pool are indexed by a non-negative integer number ranging from 0 to $n^l - 1$, so the pool $\mathbf{M}^l$ is a **lookup table** that can return corresponding kernels by giving indices. Since the kernels in the pool are indexed, $\mathbf{K}^l$ can be saved as a vector $\mathbf{C}^l$ (whose values are non-negative integers), $\mathbf{C}^l \in \mathbb{N}^{d^l}$, and $\mathbf{C}^l$ can be easily mapped to the float domain $\mathbf{K}^l$ by doing lookup operations in $\mathbf{M}^l$. Please see Figure 2.

One straightforward way to search the layer-wise pools is using brute force search based on classification performance. However, per layer, the number of possible combinations of kernels to be searched for is $(n^l)^{d^l}$ and d^l is a constant given a model. Consequently, it is not feasible to try every possible combination using brute force search as the search space is extremely large. Hence, we introduce an efficient, end-to-end searching algorithm for finding the optimal combination of kernels for making up layers and a model as follows. We utilise an auxiliary component that is a sequence of learnable temporary kernel weights $\mathbf{W}^l \in \mathbb{R}^{d^l \times k \times k}$ with the same size of $\mathbf{K}^l$. A layer needs to be filled with kernels selected from the pool and $\mathbf{W}^l$ can be regarded as placeholders. Each temporary kernel is replaced by a kernel in the pool based on a similarity metric such as $L2$ distance. Kernel weights selected from the pool form the model and are used to process inputs, resulting in classification loss. Temporary kernels are then updated by the classification loss and the similarity metric loss. We repeat this process over multiple iterations to update the temporary kernels such that an optimal set of $\mathbf{K}^l$ can be found from the pool given a new task. Temporary kernel weights can be initialised using the weights mapped from kernel indices of a previous task.

To wrap up, we search the pools for the optimal combination of kernels for each layer in the model simultaneously. The formed layers then process the input in forward propagation. However, in backpropagation, the gradients of $\mathbf{K}^l$ induced by the classification loss are used to update the temporary kernel weights $\mathbf{W}^l$, and the kernel weights in the pool remain fixed. As given in Eq. 1, searching the weights of a kernel is the nearest neighbour search problem, where $\mathbf{k}$ and $\mathbf{w}$ are the selected kernel and the temporary kernel,

respectively:

$$\mathbf{k}_i = \text{NWS}(\mathbf{w}; \mathbf{M}), \text{ where } i = \arg\min_i \|\mathbf{w} - \mathbf{k}_i\|_2 . \quad (1)$$

Since NWS is a non-differential operation, passing the gradients of selected kernels to temporary kernels is not straightforward and we use Straight-Through Estimation (STE) for this purpose. The loss can be then defined as

$$\mathcal{L}_{\text{NWS}}(x, y, \mathbf{W}, \mathbf{M})) = -\sum_{j}^{V} \delta_{y=j} \log(p_j(x)) + \|sg[\text{NWS}(\mathbf{W}; \mathbf{M})] - \mathbf{W}\|_2^2, \quad (2)$$

where the first term in Eq. 2 is a softmax cross-entropy loss, $\delta_{y=j}$ is an indicator function, and $p_j(x)$ denotes the prediction logits of the j_{th} class (V classes in total). The second term in Eq. 2 is a similarity loss (mean squared error is used in practice). Stop-gradient, $sg[.]$, is an operation that prevents gradients to propagate to its argument. As shown in Figure 2, gradients of the two aforementioned losses propagate back to the temporary kernels: (1) the direct gradients of the similarity loss; and (2) the indirect STE gradients of the classification loss.

There are a couple of points to note. For each layer, we use a separate pool. Our investigation has shown that a single shared pool for all layers performs worse than layer-wise pools. Because during the training or inference temporary kernels are not directly used for the forward propagation, they are never stored and can be discarded after the model is built. The whole training and inference process is presented in Algorithm 1.

Algorithm 1: Task Incremental Learning with Neural Weight Search

Require:
$\mathbf{C}_0$ ▷ kernel indices of the pretrained model
$\mathbf{M}$ ▷ pretrained pools
NWS ▷ neural weight search function that takes temporary kernels as inputs
$\mathcal{E}$ ▷ embedding function that encodes input kernels into corresponding non-negative integers via a pool
$\mathcal{D}$ ▷ lookup operation that returns corresponding kernels given indices via a pool (lookup table)

for *task* $t = 1; t <= N$ **do**
  get task-specific data $\mathbf{x}_t$, $\mathbf{y}_t$
  $\mathbf{W}_t \leftarrow \mathcal{D}(\mathbf{C}_{t-1}; \mathbf{M})$
  for *layer* $l = 1; l <= L$ **do**
    $\mathbf{K}_t^l$, $\text{diff}_t^l \leftarrow \text{NWS}(\mathbf{W}_t^l; M^l)$
    $\mathcal{L}_{\text{diff}} \leftarrow \mathcal{L}_{\text{diff}} + \text{diff}_t^l$
    if $l = 1$ **then**
      $\mathbf{o}_t^l \leftarrow \text{f}(\mathbf{x}_t; \mathbf{K}_t^l)$
    else
      $\mathbf{o}_t^l \leftarrow \text{f}(\mathbf{o}_t^{l-1}; \mathbf{K}_t^l)$
  $\mathcal{L}_{ce} \leftarrow min_{\mathbf{W}_t} \mathcal{L}(o_t^L, y^t)$
  $\mathbf{W}_t \leftarrow \text{UPDATE}(\mathbf{W}_t; \mathcal{L}_{ce}, \mathcal{L}_{\text{diff}})$ ▷ update temporary kernels based on Eq. 2
  $\mathbf{C}_t \leftarrow \mathcal{E}(\mathbf{W}_t; \mathbf{M})$ ▷ save model as indices
  $\mathbf{K}_t \leftarrow \mathcal{D}(\mathbf{C}_t; \mathbf{M})$ ▷ During inference, map indices back to kernel weights by indexing in pools
  Excute INFERENCE$(\mathbf{x}_t; \mathbf{K}_t)$

4.2. Design of Search Space

There could be many ways to design layer-wise search spaces (pools). One straightforward way is pretraining a model and using all kernel weights as the search space. However, in this case, as the size of this search space is proportional to the number of input and output channels of a layer, large numbers can bring about significant search costs. To tackle this, we propose a novel knowledge distillation strategy that can distil from a network a compact search space **with a predefined pool size**. Constructing the pools can be achieved by minimising the following loss function:

$$\min_{\mathbf{W},\mathbf{M}} \mathcal{L}_{\text{KP}}(\mathbf{x}, \mathbf{y}, \mathbf{W}, \mathbf{M}) = \min_{\mathbf{W}} \mathcal{L}_{\text{NWS}} + \beta \min_{\mathbf{M}} \mathcal{L}_{\text{WD}}, \quad (3)$$

where the $\mathcal{L}_{NWS}$ is the weight search loss for updating temporary kernels used in Section 4.1. We define the *weight distillation* (WD) loss as follows:

$$\mathcal{L}_{\text{WD}} = \|sg[\mathbf{W}] - \text{NWS}(\mathbf{W}; \mathbf{M})\|_2^2. \quad (4)$$

In Eq. 3, the first term optimises the temporary weights to find the optimal selection of kernel weights in the pool. The second term, i.e., weight distillation loss allows the selected kernel weights (that are trainable in this stage) from the pools to be updated, bringing them closer to corresponding temporary weights. A coefficient β is used to control the speed for updating the pools. The construction of the layer-wise pools (weight embedding space) is analogous to that of the feature embedding space discussed in VQ-VAE [33], which is trivial to implement. The pretraining and distillation are conducted simultaneously in a large-scale dataset like ImageNet [28] to ensure the generalization.

Once pretraining is completed, for each layer, the kernel weights in pools are frozen and temporary weights are discarded. The resulting pools can then be utilized for building novel models and learning new tasks as discussed in Section 4.1.

4.3. Implementation Details

Our NWS algorithm is a model-agnostic learning method, and it is implemented by simply replacing the

convolution layers (including short-cut layers) with NWS-incorporated convolution layers. An NWS-incorporated convolution layer has temporary kernels (which are discarded after searching) and a pool. Our investigation has shown that the optimal number of kernels for each pool is $n^l = 512$ (see App. 3 for details). To evaluate our approach, we present results with different architectures including ResNet-18, ResNet-34 [6], MobileNetV2 [31], and VGG [32]. For instance, for ResNet-18 [6], we replace the last fully connected layer with a convolution layer (kernel size of 1×1). The kernel size ($k \times k$) for the remaining layers is set based on the default parameters of ResNet-18.

The overall loss function in Eq. 3 is used to construct layer-wise pools by pretraining the model such as ResNet-18 on a large-scale image dataset, i.e., ImageNet [28], where we randomly initialise the kernels in the pools as well as temporary kernels. Then, during the incremental learning, layers of a new model are built simultaneously with the NWS loss (Eq. 2) in an end-to-end manner.

5. Experiments

5.1. Benchmarks

Following the previous works [20, 35], we evaluate our proposed methods on two existing TIL benchmarks: Split-CIFAR-100, and CUB-to-Sketches. The Split-CIFAR-100 benchmark contains 20 tasks, where each is a 5-way classification task with an image size of 32×32. The split is done in a way that the 5 classes in a task have the same superclass [13]. The CUB-to-Sketches benchmark contains 5 datasets, each treated as a task, including CUB-200 [34], Cars-196 [12], Flowers-102 [22], WikiArt-195 [30], and Sketches-250 [5]. Here, for instance, CUB-200 means the CUB dataset has 200 classes, and so on, and for these datasets, the images are resized to 224×224.

5.2. Evaluation Metric

We report task-wise accuracy by evaluating each task individually with the corresponding task-wise model. We also use *average accuracy* over all tasks, which is defined as: $\overline{\mathcal{A}} = \frac{1}{T}\sum_{i=1}^{T} A_i$, where T is the total number of tasks. Following [8, 20, 35] total memory cost is reported which includes the backbone model and introduced masks as well as statistics (means and variances) of batch normalizations per task for baselines or includes the shared layer-wise kernel pools and individual kernel indices as well as statistics of batch normalizations per task for our method.

5.3. Baselines

We compare our algorithm with the following baseline methods: (1) **Finetune**. It finetunes a pretrained model for each task separately. It theoretically provides the upper bound of accuracy and uncontrollable memory growth. (2) **KSM** [35]. It fixes the pretrained backbone model and learns soft kernel-wise masks for each task. (3) **PackNet** [20]. It learns binary kernel-wise masks for each task and updates the pretrained model. After each task is learned, it frees up a fixed ratio of model weights and only released weights are learnable for the next task. (4) **AQD** [4]. It quantises weights as well as features when finetuning the pretrained model in an element-wise manner for each task individually.

5.4. Experimental Setup

For a fair comparison, we use the same backbone model architecture (i.e., ResNet-18 [6]) and the same common hyperparameters for baselines and our method. For method-specific hyperparameters, default values are chosen in their original implementation (See App. 5 for details). We run each experiment with 3 different seeds and report the average results.

Pretraining. For all baselines, we used an initial ResNet-18 pretrained on ImageNet [28] (from Pytorch model zoo). For NWS, we pretrain our pools on ImageNet for 160 epochs. Each layer-wise pool has 512 kernels. Both kernels in the pools and temporary kernels are randomly initialised during the pretraining. To learn the task t, our method initialises the temporary weights from the previous task $t-1$ by looking up the indices for real-valued kernels.

Hyperparameters. Following [8, 35], all models are trained with a stochastic gradient descent (SGD) optimizer with 0.9 momentum and $1e^{-5}$ weight decay in 100 epochs. In the case of Split-CIFAR-100, the initial learning rate is set to 0.01. In the case of CUB-to-Sketches, the initial learning rate is set to 0.001. The learning rate is divided by 10 after 50 epochs and 80 epochs. For NWS, we empirically set β to 0.5 (0.1) in Eq. 3 for Split-CIFAR-100 (CUB-to-Sketches). We report further results conducted with different hyperparameters in App. 3.

Table 1: Comparison of methods described in Section 5.3 in terms of average classification accuracy and total memory cost on Split-CIFAR-100.

Method	Avg Acc (%)	Memory (MB)
Finetune	71.3	892.0
KSM [35]	71.5	192.5
PackNet [20]	67.1	55.2
AQD [4]	69.9	52.7
Ours	**73.4**	**33.9**

5.5. Experimental Results

In terms of accuracy, Figure 3 compares task-wise inference accuracy on Split-CIFAR-100, where our method

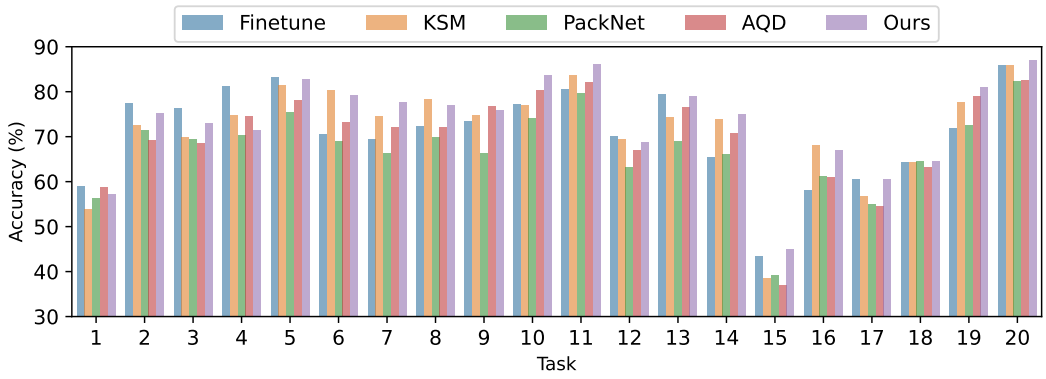

Figure 3: Comparison of methods described in Section 5.3 in terms of task-wise accuracy on Split-CIFAR-100.

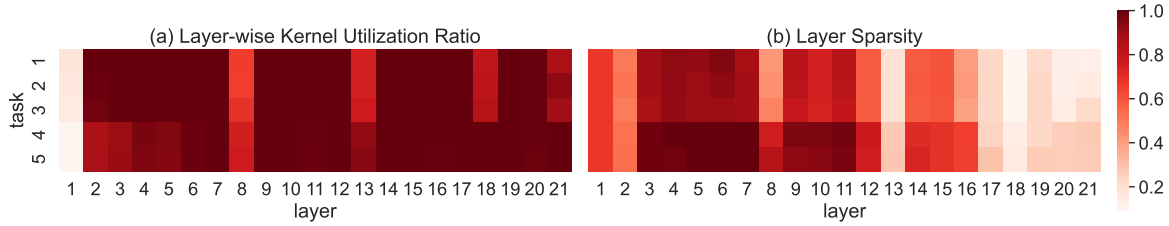

Figure 4: Heat map visualisation of (a) layer-wise kernel utilization ratio, and (b) layer sparsity obtained on the CUB-to-Sketches benchmark (denoted as task 1 to task 5 respectively).

Table 2: Comparison of methods described in Section 5.3 in terms of task-wise accuracy on CUB-to-Sketches benchmark.

Method	Task-wise Accuracy (%)						Memory (MB)
	Task1	Task2	Task3	Task4	Task5	Avg	
	CUB	Cars	Flowers	WikiArt	Sketch		
Finetune	77.4	84.1	94.5	74.2	76.9	81.4	223.0
KSM [35]	65.9	79.7	**93.5**	66.2	73.9	75.8	76.9
PackNet [20]	**78.4**	82.4	90.7	69.0	**75.3**	79.1	56.0
AQD [4]	43.0	51.8	63.4	60.5	69.7	57.7	13.7
Ours	77.0	**87.8**	93.0	**73.9**	75.1	**81.3**	**9.9**

outperforms all the methods in 8 tasks out of 20, including the Finetune. PackNet performs similarly to KSM from task 2 to 4; but for the latter tasks, its performance is worse than other baselines, supporting the phenomenon that learnable weights run out after a certain point. AQD performs slightly worse than KSM and Finetune; because network quantisation techniques limit the learning of rich representations. Looking at Table 1, our method outperforms KSM by a margin of 1.9% in terms of average accuracy. Further results on CUB-to-Sketches are presented in Table 2. PackNet achieves slightly better results than KSM, except for WikiArt. This might be due to the that the number of tasks in CUB-to-Sketches is much smaller than Split-CIFAR-100 (5 vs. 20). AQD performs much worse than others. We conjecture that it is because CUB-to-Sketches has higher resolution images as compared to Split-CIFAR-100 (224^2 vs. 32^2). Excluding Finetune which is memory-unfriendly, overall, our method achieves the best average accuracy. It performs better than other methods for the tasks, Cars and WikiArt, and is on par with the best performing baselines for the CUB (-1.4%), Flowers (-0.5%) and Sketch (-0.2%).

In terms of memory, a similar trend can be observed for both Split-CIFAR-100 and CUB-to-Sketches. Looking at the Tables 1 and 2, as compared to Finetune, PackNet, KSM and AQD, our method respectively saves up to 96%, 39% 82%, 35% on Split-CIFAR-100 benchmark and 95%, 82%, 87%, 28% memory on CUB-to-Sketches benchmark.

In terms of running time during the inference, all models excluding AQD are approximately the same. NWS can map saved kernel indices of a model to the weights of the model with negligible time thanks to the lookup operation. Taken together, our proposed method achieves competitive performance on two TIL benchmarks and demonstrates an unattainable memory saving as compared to the baselines [2]

5.6. Different Model Architectures

To demonstrate our approach can be generalised, we test NWS algorithm with another 3 different architectures: Resnet-34 [6], MobileNet-V2 [31], VGG-16 [32]. We compare NWS-incorporated models with baselines in Table 3. The baselines finetune a corresponding pretrained model (pretrained on ImageNet) for each task separately. Please refer to App. 4 for pretraining and training setups. Results show that our NWS is compatible with modern deep neural networks and NWS-incorporated networks can offer a large memory reduction with competitive inference accuracy. Note that NWS-VGG16 achieves a higher compression rate (97.5%) than that of NWS-MobileNetV2 (81%). It is because 1×1 kernels are extensively used in MobileNetV2 and from the model compression perspective for a 1×1 kernel NWS only compresses 1 float value into 1 integer value.

Table 3: Comparison of different architectures on Split-CIFAR-100.

Method	Avg Acc (%)	Memory (MB)
Finetune-Res34	77.0	1,628.0
Finetune-VGG16	75.6	1,124.0
Finetune-MobilnetV2	75.7	272.0
NWS-Res34	74.8	59.6
NWS-VGG16	74.8	28.0
NWS-MobileNetV2	71.5	52.6

5.7. Analysis of Selected Kernels

In this section, we provide further insight into how kernels are used for different layers. For this purpose, we introduce two new concepts, namely, *layer-wise kernel utilization ratio* and *layer sparsity*.

We define the *layer-wise kernel utilization ratio* (KUR) as KUR $= U^l/n^l$, where U^l is the number of unique selected kernels for the layer l and n^l is the number of kernels in the pool (512 kernels in our case). We compute KUR on the CUB-to-Sketches benchmark. The ResNet-18 has a total of 21 layers (including short-cut layers). Figure 4-(a) shows the layer 1 and 8 utilise a smaller number of unique kernels from the corresponding layer-wise pool. For layer 1 (the first layer), very low utilization ratios are observed. A larger utilization ratio indicates most of the kernels in the pool are selected. Kernels in the first layers capture coarse common local features (e.g., line, curve, and dot), whereas kernels in the following layers extract fine-grained specialized global features (e.g., ear, eye, and head). Therefore, we conjecture that diversity (large KUR) is necessary for kernels in the subsequent layers to ensure specialisation as compared to the first layers.

We also investigate the *layer sparsity*. Intuitively, the more a kernel is reused in a layer, the more important it is. A kernel selected a few times is less important in contrast; therefore, setting its weight values to zero may hardly hamper the performance, which can be used as a means for network sparsification. We denote the selection times of a unique index as h_u^l and an adaptive layer-wise threshold as $\sqrt{d^l}$ where d^l is the number of required kernels to build the layer. We formulate the layer sparsity as follows:

$$\text{LS}^l = \sum_{u=1}^{U^l} \frac{h_u^l}{d^l} \quad \text{s.t.} \quad h_u^l < \sqrt{d^l}. \tag{5}$$

Another heatmap is used to visualise the layer-wise sparsity for the CUB-to-Sketches benchmark. As shown in Figure 4 (b), we observe higher levels of layer sparsity in the latter tasks (i.e., tasks 4 and 5). It might be that the tasks are learned in a sequential manner (temporary kernels of the current task are initialised with the reconstructed kernel indices of the previous model) and sparsity might have been inherited.

6. Conclusion

In this paper, we propose a novel method called Neural Weight Search for task incremental learning. Our algorithm learns new models by searching grouped weights saved in layer-wise pools and saves learned models in the form of indices, which significantly reduces the memory cost. NWS is an out-of-the-box mechanism that can be easily integrated with modern deep learning methods. Our experiments show NWS achieves state-of-the-art performance on the Split-CIFAR-100 and CUB-to-Sketches benchmarks in terms of both accuracy and memory.

Acknowledgements

The work of Jian Jiang has been supported by the King's China Scholarship Council (K-CSC) PhD Scholarship programme and NVIDIA Academic Hardware Grant Program. The work of Oya Celiktutan has been supported by the LISI Project funded by the EPSRC UK (Grant Ref: EP/V010875/1). Finally, this work has been partially supported by Toyota Motor Europe (TME) and Toyota Motor Corporation (TMC).

[2] We also compared with CPG [8]. However, our investigation showed that if ResNet-18 is used as the backbone on Split-CIFAR100, CPG becomes sensitive to predefined thresholds that are used to control the training and tends to fail or grow too much, which led to poor performance for this method and therefore it was excluded for fairness.

References

[1] Rahaf Aljundi. Continual learning in neural networks. *arXiv preprint arXiv:1910.02718*, 2019.

[2] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014.

[3] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with a-GEM. In *International Conference on Learning Representations*, 2019.

[4] Peng Chen, Jing Liu, Bohan Zhuang, Mingkui Tan, and Chunhua Shen. Aqd: Towards accurate quantized object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 104–113, 2021.

[5] Mathias Eitz, James Hays, and Marc Alexa. How do humans sketch objects? *ACM Transactions on graphics (TOG)*, 31(4):1–10, 2012.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Saihui Hou, Xinyu Pan, Chen Change Loy, Zilei Wang, and Dahua Lin. Learning a unified classifier incrementally via rebalancing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 831–839, 2019.

[8] Ching-Yi Hung, Cheng-Hao Tu, Cheng-En Wu, Chien-Hung Chen, Yi-Ming Chan, and Chu-Song Chen. Compacting, picking and growing for unforgetting continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[9] David Isele and Akansel Cosgun. Selective experience replay for lifelong learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[10] Jian Jiang, Edoardo Cetin, and Oya Celiktutan. Ibdrr:incremental learning with information-back discrete representation replay. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 3533–3542, 2021.

[11] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences of the United States of America*, 114(13):3521–3526, 2017.

[12] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

[13] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

[14] Zhizhong Li and Derek Hoiem. Learning without Forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(12):2935–2947, dec 2018.

[15] Xialei Liu, Chenshen Wu, Mikel Menta, Luis Herranz, Bogdan Raducanu, Andrew D Bagdanov, Shangling Jui, and Joost van de Weijer. Generative feature replay for class-incremental learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 226–227, 2020.

[16] Yaoyao Liu, Bernt Schiele, and Qianru Sun. Adaptive aggregation networks for class-incremental learning. In *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, 2021.

[17] Yaoyao Liu, Yuting Su, An-An Liu, Bernt Schiele, and Qianru Sun. Mnemonics training: Multi-class incremental learning without forgetting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12245–12254, 2020.

[18] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 6468–6477, 2017.

[19] Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 67–82, 2018.

[20] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

[21] Massimiliano Mancini, Hakan Karaoguz, Elisa Ricci, Patric Jensfelt, and Barbara Caputo. Knowledge is never enough: Towards web aided deep open world recognition. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 9537–9543, 2019.

[22] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[23] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual Lifelong Learning with Neural Networks: A Review.

[24] German I Parisi, Jun Tani, Cornelius Weber, and Stefan Wermter. Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization. *Frontiers in neurorobotics*, page 78, 2018.

[25] Amal Rannen, Rahaf Aljundi, Matthew B. Blaschko, and Tinne Tuytelaars. Encoder based lifelong learning. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[26] Dushyant Rao, Francesco Visin, Andrei Rusu, Razvan Pascanu, Yee Whye Teh, and Raia Hadsell. Continual unsupervised representation learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[27] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P Lillicrap, and Greg Wayne. Experience replay for continual learning. *arXiv preprint arXiv:1811.11682*, 2018.

[28] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large

scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[29] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[30] Babak Saleh and Ahmed Elgammal. Large-scale classification of fine-art paintings: Learning the right metric on the right feature. *arXiv preprint arXiv:1505.00855*, 2015.

[31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[33] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6309–6318, Red Hook, NY, USA, 2017. Curran Associates Inc.

[34] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona. Caltech-UCSD Birds 200. Technical Report CNS-TR-2010-001, California Institute of Technology, 2010.

[35] Li Yang, Zhezhi He, Junshan Zhang, and Deliang Fan. Ksm: Fast multiple task adaption via kernel-wise soft mask learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13845–13853, 2021.

[36] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3987–3995. JMLR. org, 2017.