Abstract—Continual learning with neural networks, which aims to learn a sequence of tasks, is an important learning framework in artificial intelligence (AI). However, it often confronts three challenges: 1) overcome the catastrophic forgetting problem; 2) adapt the current network to new tasks; and 3) control its model complexity. To reach these goals, we propose a novel approach named continual learning with efficient architecture search (CLEAS). CLEAS works closely with neural architecture search (NAS), which leverages reinforcement learning techniques to search for the best neural architecture that fits a new task. In particular, we design a neuron-level NAS controller that decides which old neurons from previous tasks should be reused (knowledge transfer) and which new neurons should be added (to learn new knowledge). Such a fine-grained controller allows finding a very concise architecture that can fit each new task well. Meanwhile, since we do not alter the weights of the reused neurons, we perfectly memorize the knowledge learned from the previous tasks. We evaluate CLEAS on numerous sequential classification tasks, and the results demonstrate that CLEAS outperforms other state-of-the-art alternative methods, achieving higher classification accuracy while using simpler neural architectures.

Index Terms—Continual learning, deep learning, deep neural network, neural architecture search (NAS).

I. INTRODUCTION

In recent years, deep neural networks, such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs), have demonstrated their powerful capabilities regarding understanding implicit knowledge on various downstream tasks through their human-like learning paradigms. Meanwhile, the use of deep neural networks for sequential task learning is an interesting but challenging problem in deep learning research. Continual learning, or lifelong learning, refers to the ability to repeatedly learn new tasks and simultaneously perform well on the learned tasks. It has attracted enormous interest in artificial intelligence (AI) as it mimics a human learning process—constantly acquiring and accumulating knowledge throughout their lifetime [1], [2]. Accordingly, deploying continual learning to retrain deep neural networks helps understand the sequential tasks in new scenarios. Interestingly, it is possible to efficiently learn new tasks based on previous knowledge at relatively low computing costs. Moreover, continual learning often works with deep neural networks [3]–[5] as the flexibility in a network design can effectively allow knowledge transfer and knowledge acquisition.

Intuitively, continuous learning provides a new perspective to learn new tasks on the basis of an existing well-trained model while maintaining the predictive ability to perform multiple earlier tasks. However, continual learning with neural networks usually faces three challenges. The first one is to overcome the so-called catastrophic forgetting problem [6], which states that the network may forget what has been learned on previous tasks. The second one is to effectively adapt the current network parameters or architecture to fit a new task, and the last one is to control the network size so as not to generate an overly complex network.

In continual learning, there are two main categories of strategies that attempt to solve the aforementioned challenges. The first category is to train all the tasks within a network with a fixed capacity. For example, Rebuffi et al. [7], Lopez-Paz and Ranzato [8], and Aljundi et al. [9] replayed some old samples with the new task samples and then learned a new network from the combined training set. The drawback is that they typically require a memory system that stores past data. Kirkpatrick et al. [6] and Liu et al. [10] employed some regularization terms to prevent the reoptimized parameters from deviating too much from the previous ones. Approaches using fixed network architecture, however, cannot avoid a fundamental dilemma—they must either choose to retain good model performances on learned tasks, leaving little room for learning new tasks, or compromise the learned model performances to allow learning new tasks better.

To overcome such a dilemma, the second category is to expand the neural networks dynamically [5], [11]–[13]. They typically fix the parameters of the old neurons (partially or fully) in order to eliminate the forgetting problem and also permit adding new neurons to adapt to the learning of a new task. In general, expandable networks can achieve better model performances on all tasks than the nonexpandable ones. However, a new issue appears: expandable networks can gradually become overly large or complex, which may break the limits of the available computing resources and/or lead to overfitting.
In this article, we aim to solve the continual learning problems by proposing a new approach that only requires minimal expansion of a network so as to achieve high model performances on both learned tasks and the new task. At the heart of our approach, we leverage neural architecture search (NAS) to find a very concise architecture to fit each new task. Most notably, we design NAS to provide a neuron-level control, that is, NAS selects two types of individual neurons to compose a new architecture: 1) a subset of the previous neurons that are most useful for learning the new task and 2) a minimal number of new neurons that should be added. Reusing part of the previous neurons allows efficient knowledge transfer, and adding new neurons provides additional room for learning new knowledge. In fact, the framework we proposed is suitable for future mobile computing scenarios such as the deployment of deep neural network models on mobile devices. It is intractable to build a huge neural network model on a specific mobile device to develop continuous learning tasks due to restricted computing resources. To alleviate this challenge, our proposed framework employs a well-designed, compact, and efficient architecture that reduces the consumption of computing resources. In this article, our approach is named continual learning with efficient architecture search (CLEAS). The main features and contributions of CLEAS are given in the following.

1) CLEAS dynamically expands the network to adapt to the learning of new tasks and uses NAS to determine the new network architecture.
2) CLEAS achieves zero forgetting of the learned knowledge by keeping the parameters of the previous architecture unchanged.
3) The NAS used in CLEAS is able to provide a neuron-level control, which expands the network minimally. This leads to an effective control of the network complexity.
4) The RNN-based controller behind CLEAS is using an entire network configuration (with all neurons) as a state. This state definition deviates from the current practice in related problems that would define a state as an observation of a single neuron. Our state definition leads to improvements of 0.31%, 0.29%, and 0.75% on three benchmark datasets.
5) If the network is a convolutional network (CNN), CLEAS can even decide the best filter size that should be used in modeling the new task. The optimized filter size can further improve the model performance.

We start the rest of this article by first reviewing the related work in Section II. Then, we detail our CLEAS design in Section III. Experimental evaluations and the results are presented in Section IV.

II. RELATED WORK

In this section, we describe the related work in continual learning and NAS.

A. Continual Learning

Continual learning is often considered as an online learning paradigm where new skills or knowledge are constantly acquired and accumulated. Recently, there are remarkable advances made in many applications based on continual learning: sequential task processing [14]–[16], streaming data processing [17], self-management of resources [1], [18], and so on. A primary obstacle in continual learning, however, is the catastrophic forgetting problem and many previous works have attempted to alleviate it [19], [20]. We divide them into two categories depending on whether their networks are expandable.

The first category uses a large network with a fixed capacity. These methods try to retain the learned knowledge by either replaying old samples [7], [21], [22] or enforcing the learning with regularization terms [6], [8], [10], [23]. Sample replaying typically requires a memory system that stores old data. When learning a new task, part of the old samples are selected and added to the training data. As for regularized learning, a representative approach is elastic weight consolidation (EWC) [6], which uses the Fisher information matrix to regularize the optimization parameters, so that the important weights for previous tasks are not altered too much. Other methods, such as [8], [10], and [23], also address the optimization direction of the weights to prevent the network from forgetting the previously learned knowledge. The major limitation of using fixed networks is that it cannot properly balance the learned tasks and new tasks, resulting in either forgetting old knowledge or acquiring limited new knowledge.

To address the above issue, another stream of works proposes to dynamically expand the network, providing more room for obtaining new knowledge. For example, a progressive neural network (PGN) [12] allocates a fixed number of neurons and layers to the current model for a new task. Apparently, PGN may end up generating an overly complex network that has high redundancy and it can easily crash the underlying computing system that has very limited resources. Another approach dynamically expandable network (DEN) [13] partially mitigates the issue of PGN by using group sparsity regularization techniques. It strategically selects some old neurons to retrain and adds new neurons only when necessary. However, DEN can have a forgetting problem due to the retraining of old neurons. Another drawback is that DEN has very sensitive hyperparameters that need sophisticated tuning. Both of these algorithms only grow the network and do not have a neuron-level control, which is a significant departure from our work. Most recently, a novel method reinforced continual learning (RCL) [5] also employs NAS to expand the network, and it can further decrease the model complexity. The main difference between RCL and CLEAS is that RCL blindly reuses all the neurons from all of the previous tasks and only uses NAS to decide how many new neurons should be added. However, reusing all the old neurons has two problems. First, it creates a lot of redundancy in the new network, and some old neurons may even be misleading and adversarial. Second, excessively many old neurons reused in the new network can dominate its architecture, which may significantly limit the learning ability of the new network. Therefore, RCL does not really optimize the network architecture, and thus, it is unable to generate an efficient and effective network for learning a new task. By comparison, CLEAS designs a fine-grained NAS
that provides a neuron-level control. It optimizes every new architecture by determining whether to reuse each old neuron and how many new neurons should be added to each layer.

### B. Neural Architecture Search

NAS is another promising research topic in the AI community [24], [25]. It employs reinforcement learning techniques to automatically search for the desired network architecture for modeling a specific task. For instance, Cai et al. [26] proposed EAS to discover a superb architecture with a reinforced metacontroller that can grow the depth or width of a network; Zoph and Le [27] proposed an RNN-based controller to generate the description of a network, and the controller is reinforced by the predicting accuracy of a candidate architecture.

ENAS also has many valuable applications, such as image classification [30], [31], video segmentation [32], [33], and text representation [34]. Hence, NAS is a demonstrated powerful tool and it is especially useful in continual learning scenarios when one needs to determine what is a good architecture for the new task.

### III. Methodology

In this section, we first present an overview of the CLEAS framework, followed by a detailed description of each module. In particular, we provide the CLEAS-C module to adapt to the architecture design of CNNs. Finally, we show the detailed training algorithm of CLEAS with reinforcement learning.

#### A. Overview

There are two components in the CLEAS framework: one is the task network that continually learns a sequence of tasks and the other is the controller network that dynamically expands the task network. The two components interact with each other under the reinforcement learning context—the task network sends the controller a reward signal that reflects the performance of the current architecture design; the controller updates its policy according to the reward and then generates a new architecture for the task network to test its performance. Such interactions repeat until a good architecture is found. Fig. 1 shows the overall structure of CLEAS. On the left is the task network, depicting an optimized architecture for task \( t = 1 \) (it is using gray and pink neurons) and a candidate architecture for task \( t \). They share the same input neurons but use their own output neurons. Red circles are newly added neurons and pink ones are reused neurons from task \( t = 1 \) (or any previous task). To train the network, only the red weights that connect new–old or new–new neurons are optimized. On the right is the controller network, which implements an RNN. It provides a neuron-level control to generate a description of the task network design. Each blue square is an RNN cell that decides to use or drop a certain neuron in the task network.

#### B. NAS Model

1) **Task Network**: The task network can be any neural network with expandable ability, for example, a fully connected network or a CNN. We use the task network to model a sequence of tasks. Formally, suppose that there are \( T \) tasks and each has a training set \( D_t = \{(x_i, y_i)\}_{i=1}^{M_t} \), a validation set \( V_t = \{(x_i, y_i)\}_{i=1}^{M_t} \), and a test set \( T_t = \{(x_i, y_i)\}_{i=1}^{M_t} \), for \( t = 1, 2, \ldots, T \). We denote by \( A_t \) the network architecture that is generated to model task \( t \). Moreover, we denote \( A_t = (N_t, W_t) \) where \( N_t \) are the neurons or filters used in the network and \( W_t \) are the corresponding weights. We train the first task with a basic network \( A_1 \) by solving the standard supervised learning problem

\[
\overline{W}_1 = \arg \min_{w_t} \mathbb{L}_1(W_1; D_1)
\]

where \( \mathbb{L}_1 \) is the loss function for the first task. For the optimization procedure, we use stochastic gradient descent (SGD) with a constant learning rate. The network is trained until the required number of epochs or convergence is reached.

When task \( t \) \((t > 1) \) arrives, for every task \( k < t \), we already know its optimized architecture \( A_k \) and parameters \( \overline{W}_k \). Now, we use the controller to decide a new network architecture for task \( t \). Consider a candidate network \( A_t = (N_t, W_t) \). There are two types of neurons in \( N_t \): \( N_t^{\text{old}} \) used neurons from previous tasks and \( N_t^{\text{new}} = N_t \setminus N_t^{\text{old}} \) are the new neurons added. Based on this partition, the weights \( W_t \) can also be divided into two disjoint sets: \( W_t^{\text{old}} \) are the weights that connect old–new or new–new neurons. Formally, \( N_t^{\text{old}} = \{n \in N_t | \exists k < t \text{ s.t. } n \in N_k\} \) and \( W_t^{\text{old}} = \{w \in W_t | \exists n_1, n_2 \in N_t^{\text{old}} \text{ s.t. } \text{connects } n_1, n_2\} \). The training procedure for the new task is to only optimize the new weights \( W_t^{\text{new}} \) and leave \( W_t^{\text{old}} \) unchanged, equal to their previously optimized values \( \overline{W}_t \). Therefore, the optimization problem for the new task reads

\[
\overline{W}_t^{\text{new}} = \arg \min_{W_t^{\text{new}}} L_t(W_t^{\text{old}}, \overline{W}_t; D_t).
\]

Then, we set \( \overline{W}_t = (\overline{W}_t^{\text{old}}, \overline{W}_t^{\text{new}}) \). Finally, this candidate network \( \overline{A}_t \) with optimized weights \( \overline{W}_t \) is tested on the validation set \( \mathcal{V}_t \). The corresponding accuracy and network complexity is used to compute a reward \( R \) (described in Section III-C). The controller updates its policy based on \( R \) and generates a new candidate network \( \overline{A}_t \) to repeat the above procedure. After enough such interactions, the candidate architecture that achieves the maximal reward is the optimal one for task \( t \), i.e. \( A_t = (N_t, \overline{W}_t) \), where \( N_t \) finally denotes the neurons of the optimal architecture.

2) **Controller Network**: The goal of the controller is to provide a neuron-level control that can decide which old neurons from previous tasks can be reused and how many new neurons should be added. In our actual implementation, we assume that there is a large hypernetwork for the controller to search for a task network. Suppose that the hypernetwork has \( l \) layers and each layer \( i \) has a maximum number of \( u_i \) neurons. Each neuron has two actions, either “drop” or “use” (more actions for CNN, to be described later). Thus, the search space for the controller would be \( 2^u \), where \( n = \sum_{i=1}^{l} u_i \) is the total number of neurons. Apparently, it is infeasible to enumerate all the action combinations and determine the
best one. To deal with this issue, we treat the action sequence as a fixed-length string $a_{1:n} = a_1, a_2, \ldots, a_n$ that describes a task network. We design the controller as a long short-term memory (LSTM) network where each cell controls one $a_i$ in the hypernetwork. Formally, we denote by $\pi(a_{1:n}|s_{1:n}, \theta_c)$ the policy function of the controller network as

$$\pi(a_{1:n}|s_{1:n}, \theta_c) = P(a_1|s_{1:1}, \theta_c) \cdot P(a_2|s_{1:2}, \theta_c) \cdot \ldots \cdot P(a_n|s_{1:n}, \theta_c).$$

The state $s_{1:n}$ is a sequence that represents one state, the output is the probability of a task network described by $a_{1:n}$, and $\theta_c$ denotes the parameters of the controller network. At this point, we note that our model is a departure from those standard models where their states are individual and episodes are created by starting with different initial states (described next).

Recall that in Fig. 1, the two components in CLEAS work with each other iteratively and there are $\mathcal{H} \cdot \mathcal{U}$ such iterations where $\mathcal{H}$ is the number of episodes created and $\mathcal{U}$ denotes the length of each episode. Consider an episode $e = (s_{1:n_1}, \tilde{a}_{1:n_1}, R_1, s'_{1:n_1}, \tilde{a}_{1:n_1}, R_2, \ldots, s'_{1:n_n}, \tilde{a}_{1:n_n}, R_{n}, s_{1:n+1})$. The initial state $s_{1:n}$ is either generated randomly or copied from the terminal state $s_{1:n}$ of the previous episode. The controller starts with some initial $\theta_c$. For any $u = 1, 2, \ldots, \mathcal{U}$, the controller generates the most probable task network specified by $\tilde{a}_{1:n_u}$ from $s_{1:n_u}$ based on the following LSTM model. Concretely, we use $a_i^u = f(s_i^u, h_i^{u-1})$ as the LSTM recursion where $h_i^{u-1}$ is the hidden vector and $f$ generates $a_{1:n_u}$ from $s_{1:n_u}$. Let us point out that our RNN application $a_i^u = f(s_i^u, h_i^{u-1})$ differs from the standard practice that uses $a_i^u = f(a_i^{u-1}, h_i^{u-1})$. Action $\tilde{a}_{1:n_u}$ is optimized by selecting the most probable action string $a^u_{1:n_u}$. Note that $\tilde{a}_{1:n_u}$ specifies one candidate task network. Then, we train this task network with the associated task’s data, evaluate it on the validation set, and collect the reward $R^u$. To generate a new state $s_{1:n+1}$, we construct each $a_i^{u+1}$ from the previous action string $\tilde{a}_{1:n_u}$ and their layer index string $b_i^{u+1}$. More concretely, $s_{1:n+1} = \tilde{a}_{1:n_u} \oplus b_{1:n_u}^u$, where $\tilde{a}_{1:n_u}$, $b_{1:n_u}$ have been one-hot encoded and $\oplus$ is the concatenation operator. Finally, a new network architecture $\tilde{a}_{1:n_{u+1}}$ is generated from $s_{1:n_u}$ at the end of each episode, the controller updates its parameter $\theta_c$ by a policy gradient algorithm. After all $\mathcal{H} \cdot \mathcal{U}$ total iterations, the task network that achieves the maximum reward is used for that task.

The choice for treating the state as $s_{1:n}$ and not $s_j$ has the following two motivations. In standard NAS practices, a candidate network is sent for training every time $s_j$ is updated. This is intractable in our case as the number of neurons $n$ is typically large. For this reason, we want to train only once per $s_{1:n}$. The second reason is related and stems from the fact that the reward is given only at the level of $s_{1:n}$. For this reason, it makes sense to have $s_{1:n}$ as the state. This selection also leads to computational improvements, as shown in Section IV.

3) CLEAS-C for CNN: The design of CLEAS also works for CNN with fixed filter sizes where one filter corresponds to one neuron. However, we know that filter sizes in a CNN can have a huge impact on its classification accuracy. Therefore, we further improve CLEAS so that it can decide the best filter sizes for each task. In particular, we allow a new task to increase the filter size by one upon the previous task. For example, a filter size $3 \times 3$ used in some convolutional layer in task $t - 1$ can become $4 \times 4$ in the same layer in task $t$. Note that for one task, all the filters in the same layer must use the same filter size, but different layers can use different filter sizes.

We name the new framework as CLEAS-C. There are two major modifications to CLEAS-C. First, the output actions in the controller are now encoded by 4 bits and their meanings are “only use”, “use and extend”, “only drop” and “drop and extend” (see Fig. 2). Note that the decision of extend is made at the neuron layer, but there has to be only one decision at the layer level. To this end, we apply simple majority voting of all neurons at a layer to get the layer-level decision. The other modification regards the training procedure of the task network. The only different case we should deal with is how to optimize a filter (e.g., $4 \times 4$) that is extended from a previous smaller filter (e.g., $3 \times 3$). Our solution is to preserve the optimized parameters that are associated with the original smaller filter (the nine weights) and to only optimize the additional weights (the $16 - 9 = 7$ weights). The preserved weights are placed in the center of the larger filter, and the additional weights are initialized as the averages of their surrounding neighbor weights.

C. Training With REINFORCE

Last, we present the training procedure for the controller network. Note that each task $t$ has an independent training process, so we drop subscript $t$ here. Within an episode, each
action string $a^u_{1,n}$ represents a task network’s architecture; the network is trained and evaluated, associated with a validation accuracy as $A^u$. In addition to accuracy, we also penalize the expansion of the task network in the reward function, leading to the final reward

$$R^c = R(a^u_{1,n}) = A(a^u_{1,n}) - \alpha C(a^u_{1,n})$$  \hspace{1cm} (4)$$

where $C$ is the number of newly added neurons and $\alpha$ is a tradeoff hyperparameter. With such episodes, we train

$$J(\theta_i) = \mathbb{E}[^{\sim p(s_{x,y};\theta_i)}][R]$$  \hspace{1cm} (5)$$

by using REINFORCE. We use an exponential moving average of the previous architecture accuracies as the baseline.

We summarize the key steps of CLEAS in Algorithm 1 where $\mathcal{H}$ is the number of iterations, $U$ is the length of episodes, and $p$ is the exploration probability. We point out that we do not strictly follow the usual $\epsilon$-greedy strategy; an exploration step consists of starting an epoch from a completely random state as opposed to perturbing an existing action. In a nutshell, there exist $T$ tasks to train, and each of them contains a training set and validation set. For task $t = 1$, we first train an initial network $A_1$ on $D_1$, aiming at optimizing the weights to $\overrightarrow{W}_1$ (cf. lines 2 and 3 in Algorithm 1). For the remaining tasks, we initialize the controller parameters $\theta_i$ when tackling each of the rest tasks to ensure that each task has a different controller. For each iteration $h$ in the controller training, we first sample an exploration seed $\omega$ from the Bernoulli distribution with fixed probability $p$, that is to say, we will randomly generate a state string $s^1_{1,n}$ if $\omega = 1$. Otherwise, we set the initial state string $s^1_{1,n}$ to the last state of previous episode, i.e., $s^1_{1,n} = s^L_{1,n}$. (cf. line 13 in Algorithm 1). Next, we feed the state string into the controller and generate an action string $\overrightarrow{a}^u_{1,n}$. We will use the generated action string to configure the current task network and evaluate its performance, after which we can prepare the next state string based on the current action string and the layer encoding. Finally, we obtain the final reward according to (4) and optimize the controller via reinforcement learning. After the $H$th episode, we choose the optimal network with the highest reward as the model for the current task.

IV. EXPERIMENTS

In this section, we turn to evaluate CLEAS\textsuperscript{1} and other state-of-the-art continual learning methods on the MNIST and CIFAR-100 datasets. The key results delivered are model accuracies, network complexity, and training time. All the methods are implemented in Tensorflow and run on a GTX1080Ti GPU unit.

A. Datasets and Benchmark Algorithms

We use three benchmark datasets as follows. Each dataset is divided into $T = 10$ separate tasks. MNIST-associated tasks are trained by fully connected neural networks and CIFAR-100 tasks are trained by CNNs.

1) MNIST Permutation [6]: Ten variants of the MNIST data, where each task is transformed by a different (among tasks) and fixed (among images in the same task) permutation of pixels.

2) Rotated MNIST [5]: Another ten variants of MNIST, where each task is rotated by a different and fixed angle between $0^\circ$ and $180^\circ$.

3) Incremental CIFAR-100 [7]: The original CIFAR-100 dataset contains 60,000 $32 \times 32$ colored images that belong to 100 classes. We divide them into ten tasks and each task contains ten different classes and their data.

Each task in MNIST permutations or MNIST rotations contains 55,000 training samples, 5000 validation samples, and 10,000 test samples. Each task in CIFAR-100 contains 5000 samples for training and 1000 for testing. We randomly select 1000 samples from each task training samples as the validation samples and assure each class in a task has at least 100 validation samples. The model observes the tasks one by one and does not see any data from previous tasks. We select five other continual learning methods to compare. Note that the methods EWC and MWC use a fixed network architecture, while the other three use expandable networks.

1) EWC: It is a standard continual learning algorithm, which enforces the parameter learning in the new upcoming task to be close to the current task’s parameters [6].

\textsuperscript{1}The source code is made available at https://github.com/gcooq/CLEAS
Algorithm 1 CLEAS

Input: A sequence of tasks with training sets 
\{D_1, D_2, \ldots, D_T\}, validation sets \{V_1, V_2, \ldots, V_T\}

Output: Optimized architecture and weights for each task: \(A_t = (N_t, W_t)\) for \(t = 1, 2, \ldots, T\)

1. for \(t = 1, 2, \ldots, T\) do
2. 
3. if \(t = 1\) then
4. 
5. Train the initial network \(A_1\) on \(D_1\) with the weights optimized as \(W\);
6. else
7. Generate initial controller parameters \(\theta_c\):
8. for \(h = 1, 2, \ldots, H\) do
9. 
10. /* A new episode */
11. \(w \sim \text{Bernoulli}(p)\);
12. if \(w = 1\) or \(h = 1\) then
13. 
14. /* Exploration */
15. Generate a random state string \(s_1^{\text{fn}}\) but keep layer encodings fixed;
16. else
17. Set initial state string \(s_1^{\text{fn}} = s_{h-1}^{\text{fn+1}}\), i.e. the last state of previous episode \((h - 1)\);
18. end
19. for \(u = 1, 2, \ldots, U\) do
20. Generate the most probable action string \(\tilde{a}_u^{\text{fn}}\) from \(s_1^{\text{fn}}\) by the controller;
21. Configure the task network as \(A_u\) based on \(\tilde{a}_u^{\text{fn}}\) and train weights \(W_u\) on \(D_r\);
22. Evaluate \(A_u\) with trained \(W_u\) on \(V_l\) and compute reward \(R_u\);
23. Construct \(s_{1,n}^{\text{fn}}\) from \(a_1^{\text{fn}}\) and \(b_1^{\text{fn}}\) where \(b_1^{\text{fn}}\) is the layer encoding;
24. end
25. Update \(\theta_c\) by REINFORCE using \((s_1^{\text{fn}}, \tilde{a}_1^{\text{fn}}, R_1, \ldots, s_U^{\text{fn}}, \tilde{a}_U^{\text{fn}}, R_U, s_{1,n}^{U+1})\);
26. Store \(A_h = (N_h, W_h)\) where \(u = \arg \max_u R_u\) and \(R_h = \max_u R_u\);
27. Store \(A_t = A_h\) where \(h = \arg \max_h R_h\);
28. end
29. end
30. end

2) MWC: It is an extension of EWC. By assuming some extent of correlation between consecutive tasks, it uses regularization terms to prevent large deviation of the network weights when reoptimized [23].

3) PGN: It expands the task network by adding a fixed number of neurons and layers [12].

4) DEN: It dynamically decides the number of new neurons by performing selective retraining and network split [13].

5) RCL: It uses NAS to decide the number of new neurons. It completely eliminates the forgetting problem by holding the previous neurons and their weights unchanged [5].

For the two MNIST datasets, we follow [5] to use a three-layer fully connected network. We start with 784-312-128-10 neurons with RELU activation for the first task. For CIFAR-100, we develop a modified version of LeNet [35] that has three convolutional layers and three fully connected layers. We start with 16 filters in each layer with sizes of 3 × 3, 3 × 3, and 4 × 4 and stride of 1 per layer. Besides, to fairly compare the network choice with [5] and [13], we set: \(u_1 = 1000\) for MNIST and \(u_t = 128\) for CIFAR-100. We also use \(H = 200\) and \(U = 1\). Since the action in CLEAS depends on the previous state, we generate random actions with an initial probability (exploration probability) to avoid local minima. To this end, the exploration probability \(p\) is set to be 30%.

We also implement a version with states corresponding to individual neurons where the controller is following \(a_{j-1}^u, h_{j-1}^u\). We configure this version under the same experimental settings as CLEAS and test it on the three datasets. The results show that compared to CLEAS, this version exhibits an inferior performance of \(-0.31\%\), \(-0.29\%\), and \(-0.75\%\) in relative accuracy on the three datasets. Details can be found in Section IV-D.

B. Experimental Results

1) Model Accuracy: We first report the averaged model accuracies among all tasks. Fig. 3 shows the relative improvements of the network-expandable methods against EWC (numbers on the top are their absolute accuracies). We clearly
observe that methods with expandability can achieve much better performance than MWC. Furthermore, we see that CLEAS outperforms other methods. The average relative accuracy improvement of CLEAS versus RCL (the state-of-the-art method and the second best performer) is 0.21%, 0.21%, and 6.70%. There are two reasons: 1) we completely overcome the forgetting problem by not altering the old neurons/filters and 2) our neuron-level control can precisely pick useful old neurons as well as new neurons to better model each new task.

2) Network Complexity: Besides model performance, we also care about how complex the network is when used to model each task. With possibly limited computing resources and the need for continual learning, CLEAS aims at providing a simpler network for new task training by selectively reusing some neurons from earlier tasks, meanwhile adding some new neurons. We thus report the average number of model weights across all tasks in Fig. 4. First, no surprise to see that MWC consumes the least number of weights since its network is nonexpandable. However, this also limits its model performance. We also find that PGN, DEN, RCL, and CLEAS tend to use more parameters because of the expandable ability. Second, we observe that our CLEAS uses the least number of weights compared to the other four methods. The average relative complexity improvement of CLEAS versus RCL is 29.9%, 19.0%, and 51.0% reduction. RCL uses the conventional reinforcement learning method to define states and actions for policy optimization. Actually, the task network in the RCL model can always be expanded, whereby the new task training inherits a part of the benefits from the expanded network. However, CLEAS under specially designed reinforcement learning pays more attention to finding a better subarchitecture from both old and new neurons because of its neuron-level control policy. Hence, it supports the fact that our NAS using neuron-level control can find a very efficient architecture to model every new task.

3) Network Descriptions: We visualize some examples of network architectures generated by our controller. Fig. 5 shows four optimal configurations (tasks 2–5) of the CNN used to model CIFAR-100. Each task uses three convolutional layers and each square represents a filter. A white square means that it is not used by the current task, a red square represents that it was trained by some earlier task and now reused by the current task, a light yellow square means that it was trained before but not reused, and a dark yellow square depicts a new filter added. According to the figure, we note that CLEAS tends to maintain a concise architecture all the time. As the task index increases, it drops more old filters and only reuses a small portion of them that are useful for current task training, and it is adding fewer new neurons.

4) CLEAS-C: We also test CLEAS-C which decides the best filter sizes for CNNs. In the CIFAR-100 experiment, CLEAS uses fixed filter sizes $3 \times 3$, $3 \times 3$, and $4 \times 4$ in its three convolutional layers. By comparison, CLEAS-C starts with the same sizes but allows each task to increase the sizes by one. The results show that the final sizes become $4 \times 4$, $8 \times 8$, and $8 \times 8$ after training the ten tasks with CLEAS-C. As shown in Fig. 6, it achieves a much higher accuracy of 67.4% than CLEAS (66.9%), i.e., a 0.7% improvement. It suggests that customized filter sizes can better promote model performances. On the other hand, the complexity of CLEAS-C increases by 92.6%.

5) Neuron Allocation: We compare CLEAS to RCL on neuron reuse and neuron allocation. Fig. 7 visualizes the number of reused neurons (yellow and orange for RCL and pink and red for CLEAS) and new neurons (dark blue for both methods). There are two observations. On the one hand, CLEAS successfully drops many old neurons that are redundant or useless, ending up maintaining a much simpler network. On the other hand, we observe that both of the methods recommend a similar number of new neurons for each task. Therefore, the superiority of CLEAS against RCL lies more on its selection of old neurons. RCL blindly reuses all previous neurons.

6) Training Time: We also report the training time in Fig. 8. It is as expected that CLEAS’ running time is on the high end due to the neuron-level control that results in using a much longer RNN for the controller. On the positive note, the increase in the running time is not substantial. Specifically, CLEAS aims at discovering an optimal substructure from the trained neurons. Besides, it can make a reinforced choice by combining additional new neurons for new task knowledge learning. We use the RNN as our controller network where each output determines the probability of the corresponding task network neuron being selected. Compared to the best baseline RCL, the length of our RNN is longer than RCL (longer input sequences). However, our controller is simpler than the one used by RCL since it uses drastically fewer parameters. For example, the output size of the controller for MNIST consists of two digits, while the output size in RCL is much larger. Therefore, we observe that the increase in the running time is not substantial. Besides, as shown in Figs. 3 and 4, CLEAS selects a much simpler task network architecture for task training and nevertheless obtains better test results.

7) Variance of Random Seed: In this section, we study the variances of the accuracy results conducted with different random seeds. As shown in Fig. 9, we use five different seeds to initialize our CLEAS and report the averaged model accuracy among all tasks. We observe that all of the results with different seeds outperform the best benchmark RCL, which demonstrates that our proposed CLEAS is robust when using different seeds.

8) Hyperparameter Sensitivity: Now, we turn to show the hyperparameter analysis. To elaborate the impact of
Fig. 5. Examples of CNN architectures for CIFAR-100.

Fig. 7. Neuron allocation for MNIST permutation.

Fig. 8. Training time.

hyperparameter we used in CLEAS, we select two significant hyperparameters to illustrate the sensitivity to model performance. The observation is that the hyperparameters used in CLEAS are not as sensitive as those of DEN and RCL. Under all hyperparameter settings, CLEAS performs the best. In detail, Fig. 10 shows the sensitivity of hyperparameter $\alpha$ in (4). We can see the clear tradeoff between model performance and complexity. The best choice of $\alpha$ for MNIST is between $[10^{-4}, 10^{-3}]$ where the network is simpler but preserves good performance as well. For CIFAR-100, $\alpha$ should be between $[10^{-3}, 10^{-2}]$. In Fig. 11, we vary another hyperparameter that is the maximum number of new neurons that can be allocated per layer to a new task. As expected, as the maximum number increases, the overall model performances raise as well. However, we see that CLEAS always achieves the highest accuracy under different settings.

Fig. 9. Impact of random seed. (a) MNIST permutation. (b) CIFAR-100.

Fig. 10. Sensitivity of $\alpha$ value. (a) MNIST permutation. (b) CIFAR-100.

Fig. 11. Maximum number of new neurons. (a) MNIST permutation. (b) CIFAR-100.

To evaluate the impact of exploration probability $p$, we set different probabilities, spanning from 0 to 1, to observe the variances of testing results. As shown in Fig. 12, the value of exploration probability does affect the optimization of our controller, especially for the CIFAR-100 dataset. Thus, we should meticulously select a suitable value of exploration probability.

Finally, we report the impacts of hidden size and initial learning rate in the controller network. As shown in Figs. 13 and 14, the results illustrate that the larger hidden
size or learning rate does not bring higher accuracy, which further indicates that the small controller is capable of achieving promising results while bringing the decrease of the model complexity.

9) **Hyperparameter Importance:** In addition to the above, we also investigate the hyperparameter importance regarding our designed controller network. In detail, we are inspired by [36] and use the grid search method to generate a set of hyperparameters according to Table I. We regard each of the hyperparameters as a metafeature and the testing accuracy as the ground truth. Then, we use the popular random forest method with permutation importance (PI) to evaluate the contribution of each metafeature (i.e., hyperparameter). As shown in Fig. 15, the closer to the top [e.g., Fig. 15(c)] or right [e.g., Fig. 15(d)] of the figure, the more important the hyperparameters are. We clearly find that the maximum number of new neurons can be assigned to each task and $\alpha$ is more important than others, followed by hidden size $h_i$ and exploration probability $p$.

C. Discussions

In CLEAS, as new sequential tasks require the task network adjustments for training, each new task only needs a small subset of old neurons after receiving a network description from the controller. We evaluate CLEAS as well as other alternative methods on numerous sequential classification tasks. The results lend great credence to the fact that CLEAS is able to achieve higher classification accuracy while using simpler neural architectures. Compared to the state-of-the-art method RCL [5], we improve the model accuracy relatively by 0.21%, 0.21%, and 6.70% on the three benchmark datasets and reduce network complexity by 29.9%, 19.0%, and 51.0%, respectively.

Here, we experimentally compare CLEAS to the standard implementation of an NAS controller that considers each output of an RNN-based network as an individual action and it starts with only one initialized state $s_0$, as shown in Fig. 16. Therefore, such a controller considers $(s_0, a_0, 0, s_1, 0, \ldots, s_{n-1}, a_{n-1}, R)$ as one episode where $s_j = a_{j-1}$, and the real reward $R$ is given only after all states and actions are played. However, the controller of CLEAS considers a sequence of candidate task networks as one episode, and each candidate receives a reward immediately, that is, CLEAS considers $(s_{1,1}, a_{1,1}, R_1, s_{1,2}, a_{1,2}, R_2, \ldots, s_{l,1}, a_{l,1}, R_l, s_{l,2})$ as one episode (recall this from Section III-B).

1) **CLEAS Versus Standard NAS Controller:** We evaluate these two versions on the same three datasets that were used in the above. Fig. 17 shows each task accuracy of the three datasets. We find that the controller implemented in the standard way achieves inferior model performances.
Fig. 16. Standard implementation of NAS controller.

Fig. 17. Task accuracy of standard NAS controller versus CLEAS NAS controller. (a) MNIST permutation. (b) Rotated MNIST. (c) CIFAR-100.

96.0%, 96.7%, and 66.4% in average accuracy on the three datasets. By comparison, CLEAS achieves 96.3%, 97.0%, and 66.9%, thus yielding 0.31%, 0.29%, and 0.75% relative improvement. Besides, we also implement a random search version for the task network search by using the same number of models as our CLEAS. Such simple version achieves 95.1%, 96.3%, and 62.7% on three datasets, which is actually worse than our CLEAS and standard NAS controller. Since the simple version still has a probability to extend the task network by adding new neurons, it can keep active in training for new coming task. However, the random search from old neurons might destroy the inherent information for old tasks.

V. CONCLUSION

We have proposed and developed a novel approach CLEAS to tackle continual learning problems. CLEAS is a network-expandable approach that uses NAS to dynamically determine the optimal architecture for each task. NAS is able to provide a neuron-level control that decides the reuse of old neurons and the number of new neurons needed. Such a fine-grained control can maintain a very concise network through all tasks. Also, we completely eliminate the random search version for the task network search by using the same number of models as our CLEAS and standard NAS controller. Since the simple version still has a probability to extend the task network by adding new neurons, it can keep active in training for new coming task. However, the random search from old neurons might destroy the inherent information for old tasks.

REFERENCES

[1] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, “Continual lifelong learning with neural networks: A review,” Neural Netw., vol. 113, pp. 54–71, May 2019.
[2] A. Ororbia, A. Mali, C. L. Giles, and D. Kifer, “Continual learning of recurrent neural networks by locally aligning distributed representations,” IEEE Trans. Neural Netw. Learn. Syst., vol. 31, no. 10, pp. 4267–4278, Oct. 2020.
[3] K. Javed and M. White, “Meta-learning representations for continual learning,” in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 1818–1828.
[4] C. V. Nguyen, Y. Li, T. D. Bui, and R. E. Turner, “Variational continual learning,” 2017, arXiv:1710.10628.
[5] J. Xu and Z. Zhu, “Reinforced continual learning,” in Proc. Adv. Neural Inf. Process. Syst., 2018, pp. 899–908.
[6] K. James et al., “Overcoming catastrophic forgetting in neural networks,” Proc. Nat. Acad. Sci. USA, vol. 114, no. 13, pp. 3521–3526, Mar. 2017.
[7] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, “ICaRL: Incremental classifier and representation learning,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), Jul. 2017, pp. 2001–2010.
[8] D. Lopez-Paz and M. Ranzato, “Gradient episodic memory for continual learning,” in Proc. Adv. Neural Inf. Process. Syst., 2017, pp. 6467–6476.
[9] R. Aljundi, F. Babiloni, M. Elhoseiny, M. Rohrbach, and T. Tuylraels, “Memory aware synapses: Learning what (not) to forget,” in Proc. Eur. Conf. Comput. Vis., 2018, pp. 139–154.
[10] X. Liu, M. Masana, L. Herranz, J. Van de Weijer, A. M. Lopez, and A. D. Bagdanov, “Rotate your networks: Better weight consolidation and less catastrophic forgetting,” in Proc. 24th Int. Conf. Pattern Recognit. (ICPR), Aug. 2018, pp. 2262–2268.
[11] H. Li, P. Barnaghi, S. Enshaeifar, and F. Ganz, “Continual learning using Bayesian neural networks,” IEEE Trans. Neural Netw. Learn. Syst., vol. 32, no. 9, pp. 4243–4252, Sep. 2021.
[12] A. A. Rusu et al., “Progressive neural networks,” 2016, arXiv:1606.04671.
13] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, “Lifelong learning with dynamically expandable networks,” in Proc. Int. Conf. Learn. Represent., 2018, pp. 1–11.
14] S. Thrun, “A lifelong learning perspective for mobile robot control,” in Intelligent Robots and Systems. Amsterdam, The Netherlands: Elsevier, 1995, pp. 201–214.
15] H. Li, W. Dong, and B.-G. Hu, “Incremental concept learning via online generative memory recall,” IEEE Trans. Neural Netw. Learn. Syst., vol. 32, no. 7, pp. 3206–3216, Jul. 2021.
16] G. Sun, Y. Cong, Y. Zhang, G. Zhao, and Y. Fu, “Continual multiview task learning via deep matrix factorization,” IEEE Trans. Neural Netw. Learn. Syst., vol. 32, no. 1, pp. 139–150, Jan. 2021.

17] R. Aljundi, K. Kelchtermans, and T. Tuytelaars, “Task-free continual learning,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2019, pp. 11254–11263.
18] T. Diethe, T. Borchert, E. Thereska, B. Balle, and N. Lawrence, “Continual learning in practice,” 2019, arXiv:1903.05202.
19] G. Sun, Y. Cong, Q. Wang, B. Zhong, and Y. Fu, “Representative task self-selection for flexible clustered lifelong learning,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR). Washington, DC, USA.

20] Q. Lao, X. Jiang, M. Havaei, and Y. Bengio, “A two-stream continual learning system with variational domain-agnostic feature replay,” IEEE Trans. Neural Netw. Learn. Syst., early access, Mar. 3, 2021, doi: 10.1109/TNNLS.2021.3075453.
21] D. Rolnick, A. Ahuja, J. Schwartz, T. Lillicrap, and G. Wayne, “Experience replay for continual learning,” in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 348–358.

22] A. Robins, “Catastrophic forgetting, rehearsal and pseudorehearsal,” Connection Sci., vol. 7, no. 2, pp. 123–146, 1995.
23] J. Zhang et al., “Regularize, expand and compress: NonExpansive continual learning,” in Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV), Mar. 2020, pp. 854–862.
24] T. Elsken, J. H. Metzen, and F. Hutter, “Neural architecture search: A survey,” J. Mach. Learn. Res., vol. 20, no. 55, pp. 1–21, 2019.
25] X. Gong, S. Chang, Y. Jiang, and Z. Wang, “AutoGAN: Neural architecture search for generative adversarial networks,” in Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV), Oct. 2019, pp. 3224–3234.

26] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, “Efficient architecture search by network transformation,” in Proc. 32nd AAAI Conf. Artif. Intell., 2018, pp. 2787–2794.
27] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” 2016, arXiv:1611.01578.
28] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, “Efficient neural architecture search via parameters sharing,” in Int. Conf. Mach. Learn., 2018, pp. 4095–4104.
29] K. Yu, R. Ranftl, and M. Salzmann, “An analysis of super-net heuristics in weight-sharing NAS,” IEEE Trans. Pattern Anal. Mach. Intell., early access, Aug. 30, 2021, doi: 10.1109/TPAMI.2021.3108480.
30] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized evolution for image classifier architecture search,” in Proc. AAAI Conf. Artif. Intell., vol. 33, pp. 4780–4789.
31] I. Radosavovic, J. Johnson, S. Xie, W.-Y. Lo, and P. Dollár, “On network design spaces for visual recognition,” in Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV), Oct. 2019, pp. 1882–1890.
32] V. Nekrasov, H. Chen, C. Shen, and I. Reid, “Architecture search of dynamic cells for semantic video segmentation,” in Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV), Mar. 2020, pp. 1970–1979.
33] P. Lin, P. Sun, G. Cheng, S. Xie, X. Li, and J. Shi, “Graph-guided architecture search for real-time semantic segmentation,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2020, pp. 4203–4212.
34] Y. Wang et al., “TextNAS: A neural architecture search space tailored for text representation,” 2019, arXiv:1912.10729.
35] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proc. IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
36] J. N. van Rijn and F. Hutter, “Hyperparameter importance across datasets,” in Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Jul. 2018, pp. 2367–2376.