CHAPTER 4

Continual Learning and Catastrophic Forgetting

In the recent years, lifelong learning (LL) has attracted a great deal of attention in the deep learning community, where it is often called *continual learning*. Though it is well-known that deep neural networks (DNNs) have achieved state-of-the-art performances in many machine learning (ML) tasks, the standard multi-layer perceptron (MLP) architecture and DNNs suffer from *catastrophic forgetting* [McCloskey and Cohen, 1989] which makes it difficult for continual learning. The problem is that when a neural network is used to learn a sequence of tasks, the learning of the later tasks may degrade the performance of the models learned for the earlier tasks. Our human brains, however, seem to have this remarkable ability to learn a large number of different tasks without any of them negatively interfering with each other. Continual learning algorithms try to achieve this same ability for the neural networks and to solve the catastrophic forgetting problem. Thus, in essence, continual learning performs incremental learning of new tasks. Unlike many other LL techniques, the emphasis of current continual learning algorithms has not been on how to leverage the knowledge learned in previous tasks to help learn the new task better. In this chapter, we first give an overview of catastrophic forgetting (Section 4.1) and survey the proposed continual learning techniques that address the problem (Section 4.2). We then introduce several recent continual learning methods in more detail (Sections 4.3–4.8). Two evaluation papers are also covered in Section 4.9 to evaluate the performances of some existing continual learning algorithms. Last but not least, we give a summary of the chapter and list the relevant evaluation datasets.

4.1 CATASTROPHIC FORGETTING

Catastrophic forgetting or *catastrophic interference* was first recognized by McCloskey and Cohen [1989]. They found that, when training on new tasks or categories, a neural network tends to forget the information learned in the previous trained tasks. This usually means a new task will likely override the weights that have been learned in the past, and thus degrade the model performance for the past tasks. Without fixing this problem, a single neural network will not be able to adapt itself to an LL scenario, because it *forgets* the existing information/knowledge when it learns new things. This was also referred to as the stability-plasticity dilemma in Abraham and Robins [2005]. On the one hand, if a model is too stable, it will not be able to consume new information from the future training data. On the other hand, a model with sufficient plasticity

suffers from large weight changes and forgets previously learned representations. We should note that catastrophic forgetting happens to traditional multi-layer perceptrons as well as to DNNs. Shadow single-layered models, such as self-organizing feature maps, have been shown to have catastrophic interference too [Richardson and Thomas, 2008].

A concrete example of catastrophic forgetting is transfer learning using a deep neural network. In a typical transfer learning setting, where the source domain has plenty of labeled data and the target domain has little labeled data, *fine-tuning* is widely used in DNNs [Dauphin et al., 2012] to adapt the model for the source domain to the target domain. Before fine-tuning, the source domain labeled data is used to pre-train the neural network. Then the output layers of this neural network are retrained given the target domain data. Backpropagation-based fine-tuning is applied to adapt the source model to the target domain. However, such an approach suffers from catastrophic forgetting because the adaptation to the target domain usually disrupts the weights learned for the source domain, resulting inferior inference in the source domain.

Li and Hoiem [2016] presented an excellent overview of the traditional methods for dealing with catastrophic forgetting. They characterized three sets of parameters in a typical approach:

- θ_s: set of parameters shared across all tasks;
- θ_o: set of parameters learned specifically for previous tasks; and
- θ_n: randomly initialized task-specific parameters for new tasks.

Li and Hoiem [2016] gave an example in the context of image classification, in which θ_s consists of five convolutional layers and two fully connected layers in the AlexNet architecture [Krizhevsky et al., 2012]. θ_o is the output layer for classification [Russakovsky et al., 2015] and its corresponding weights. θ_n is the output layer for new tasks, e.g., scene classifiers.

There are three traditional approaches to learning θ_n with knowledge transferred from θ_s.

- **Feature Extraction** (e.g., Donahue et al. [2014]): both θ_s and θ_o remain the same while the outputs of some layers are used as features for training θ_n for the new task.
- **Fine-tuning** (e.g., Dauphin et al. [2012]): θ_s and θ_n are optimized and updated for the new task while θ_o remains fixed. To prevent large shift in θ_s, a low learning rate is typically applied. Also, for the similar purpose, the network can be *duplicated and fine-tuned* for each new task, leading to N networks for N tasks. Another variation is to fine-tune parts of θ_s, for example, the top layers. This can be seen as a compromise between fine-tuning and feature extraction.
- **Joint Training** (e.g., Caruana [1997]): All the parameters θ_s, θ_o, θ_n are jointly optimized across all tasks. This requires storing all the training data of all tasks. Multi-task learning (MTL) typically takes this approach.

The pros and cons of these methods are summarized in Table 4.1. In light of these pros and cons, Li and Hoiem [2016] proposed an algorithm called "Learning without Forgetting" that explicitly deals with the weaknesses of these methods; see Section 4.3.

Table 4.1: Summary of traditional methods for dealing with catastrophic forgetting. Adapted from Li and Hoiem [2016].

Category	Feature Extraction	Fine-Tuning	Duplicate and Fine-Tuning	Joint Training
New task performance	Medium	Good	Good	Good
Old task performance	Good	Bad	Good	Good
Training efficiency	Fast	Fast	Fast	Slow
Testing efficiency	Fast	Fast	Slow	Fast
Storage requirement	Medium	Medium	Large	Large
Require previous task data	No	No	No	Yes

4.2 CONTINUAL LEARNING IN NEURAL NETWORKS

A number of continual learning approaches have been proposed to lessen catastrophic forgetting recently. This section gives an overview for these newer developments. A comprehensive survey on the same topic is also given in Parisi et al. [2018a].

Much of the existing work focuses on *supervised learning* [Parisi et al., 2018a]. Inspired by fine-tuning, Rusu et al. [2016] proposed a progressive neural network that retains a pool of pretrained models and learns lateral connections among them. Kirkpatrick et al. [2017] proposed a model called Elastic Weight Consolidation (EWC) that quantifies the importance of weights to previous tasks, and selectively adjusts the plasticity of weights. Rebuffi et al. [2017] tackled the LL problem by retaining an exemplar set that best approximates the previous tasks. A network of experts is proposed by Aljundi et al. [2016] to measure task relatedness for dealing with catastrophic forgetting. Rannen Ep Triki et al. [2017] used the idea of autoencoder to extend the method in "Learning without Forgetting" [Li and Hoiem, 2016]. Shin et al. [2017] followed the Generative Adversarial Networks (GANs) framework [Goodfellow, 2016] to keep a set of generators for previous tasks, and then learn parameters that fit a mixed set of real data of the new task and replayed data of previous tasks. All these works will be covered in details in the next few sections.

Instead of using knowledge distillation as in the model "Learning without Forgetting" (LwF) [Li and Hoiem, 2016], Jung et al. [2016] proposed a less-forgetful learning that regularizes the final hidden activations. Rosenfeld and Tsotsos [2017] proposed controller modules to optimize loss on the new task with representations learned from previous tasks. They found

that they could achieve satisfactory performance while only requiring about 22% of parameters of the fine-tuning method. Ans et al. [2004] designed a dual-network architecture to generate pseudo-items which are used to self-refresh the previous tasks. Jin and Sendhoff [2006] modeled the catastrophic forgetting problem as a multi-objective learning problem and proposed a multi-objective pseudo-rehearsal framework to interleave base patterns with new patterns during optimization. Nguyen et al. [2017] proposed variational continual learning by combining online variational inference (VI) and Monte Carlo VI for neural networks. Motivated by EWC [Kirkpatrick et al., 2017], Zenke et al. [2017] measured the synapse consolidation strength in an online fashion and used it as regularization in neural networks. Seff et al. [2017] proposed to solve continual generative modeling by combining the ideas of GANs [Goodfellow, 2016] and EWC [Kirkpatrick et al., 2017].

Apart from regularization-based approaches mentioned above (e.g., LwF [Li and Hoiem, 2016], EWC [Kirkpatrick et al., 2017]), dual-memory-based learning systems have also been proposed for LL. They are inspired by the complementary learning systems (CLS) theory [Kumaran et al., 2016, McClelland et al., 1995] in which memory consolidation and retrieval are related to the interplay of the mammalian hippocampus (short-term memory) and neocortex (long-term memory). Gepperth and Karaoguz [2016] proposed using a modified self-organizing map (SOM) as the long-term memory. To complement it, a short-term memory (STM) is added to store novel examples. During the sleep phase, the whole content of STM is replayed to the system. This process is known as intrinsic replay or pseudo-rehearsal [Robins, 1995]. It trains all the nodes in the network with new data (e.g., from STM) and replayed samples from previously seen classes or distributions on which the network has been trained. The replayed samples prevents the network from forgetting. Kemker and Kanan [2018] proposed a similar dual-memory system called FearNet. It uses a hippocampal network for STM, a medial prefrontal cortex (mPFC) network for long-term memory, and a third neural network to determine which memory to use for prediction. More recent developments in this direction include Deep Generative Replay [Shin et al., 2017], DGDMN [Kamra et al., 2017] and Dual-Memory Recurrent Self-Organization [Parisi et al., 2018b].

Some other related works include Learn++ [Polikar et al., 2001], Gradient Episodic Memory [Lopez-Paz et al., 2017], Pathnet [Fernando et al., 2017], Memory Aware Synapses [Aljundi et al., 2017], One Big Net for Everything [Schmidhuber, 2018], Phantom Sampling [Venkatesan et al., 2017], Active Long Term Memory Networks [Furlanello et al., 2016], Conceptor-Aided Backprop [He and Jaeger, 2018], Gating Networks [Masse et al., 2018, Serrà et al., 2018], PackNet [Mallya and Lazebnik, 2017], Diffusion-based Neuromodulation [Velez and Clune, 2017], Incremental Moment Matching [Lee et al., 2017b], Dynamically Expandable Networks [Lee et al., 2017a], and Incremental Regularized Least Squares [Camoriano et al., 2017].

There are some *unsupervised learning* works as well. Goodrich and Arel [2014] studied unsupervised online clustering in neural networks to help mitigate catastrophic forgetting. They

proposed building a path through the neural network to select neurons during the feed-forward pass. Each neural is assigned with a cluster centroid, in addition to the regular weights. In the new task, when a sample arrives, only the neurons whose cluster centroid points are close to the sample are selected. This can be viewed as a special dropout training [Hinton et al., 2012]. Parisi et al. [2017] tackled LL of action representations by learning unsupervised visual representation. Such representations are incrementally associated with action labels based on occurrence frequency. The proposed model achieves competitive performance compared to models trained with predefined number of action classes.

In the *reinforcement learning* applications [Ring, 1994], other than the works mentioned above (e.g., Kirkpatrick et al. [2017], Rusu et al. [2016]), Mankowitz et al. [2018] proposed a continual learning agent architecture called Unicorn. The Unicorn agent is designed to have the ability to simultaneously learn about multiple tasks including the new tasks. The agent can reuse its accumulated knowledge to solve related tasks effectively. Last but not least, the architecture aims to aid agent in solving tasks with deep dependencies. The essential idea is to learn multiple tasks off-policy, i.e., when acting on-policy with respect to one task, it can use this experience to update policies of related tasks. Kaplanis et al. [2018] took the inspiration from biological synapses and incorporated different timescales of plasticity to mitigate catastrophic forgetting over multiple timescales. Its idea of synaptic consolidation is along the lines of EWC [Kirkpatrick et al., 2017]. Lipton et al. [2016] proposed a new reward shaping function that learns the probability of imminent catastrophes. They named it as *intrinsic fear*, which is used to penalize the Q-learning objective.

Evaluation frameworks were also proposed in the context of catastrophic forgetting. Goodfellow et al. [2013a] evaluated traditional approaches including dropout training [Hinton et al., 2012] and various activation functions. More recent continual learning models were evaluated in Kemker et al. [2018]. Kemker et al. [2018] used large-scale datasets and evaluated model accuracy on both old and new tasks in the LL setting. See Section 4.9 for more details. In the next few sections, we discuss some representative continual learning approaches.

4.3 LEARNING WITHOUT FORGETTING

This section describes the approach called *Learning without Forgetting* given in Li and Hoiem [2016]. Based on the notations in Section 4.1, it learns θ_n (parameters for the new task) with the help of θ_s (shared parameters for all tasks) and θ_o (parameters for old tasks) without degrading much of the performance on the old tasks. The idea is to optimize θ_s and θ_n on the new task with the constraint that the predictions on the new task's examples using θ_s and θ_o do not shift much. The constraint makes sure that the model still "remembers" its old parameters, for the sake of maintaining satisfactory performance on the previous tasks.

The algorithm is outlined in Algorithm 4.1. Line 2 records the predictions Y_o of the new task's examples X_n using θ_s and θ_o, which will be used in the objective function (Line 7). For each new task, nodes are added to the output layer, which is fully connected to the layer beneath.

These new nodes are first initialized with random weights θ_n (Line 3). There are three parts in the objective function in Line 7.

Algorithm 4.1 Learning without Forgetting

Input: shared parameters θ_s, task-specific parameters for old tasks θ_o, training data X_n, Y_n for the new task.
Output: updated parameters θ_s^*, θ_o^*, θ_n^*.

1: // Initialization phase.
2: $Y_o \leftarrow \text{CNN}(X_n, \theta_s, \theta_o)$
3: $\theta_n \leftarrow \text{RANDINIT}(|\theta_n|)$
4: // Training phase.
5: Define $\hat{Y}_n \equiv \text{CNN}(X_n, \hat{\theta}_s, \hat{\theta}_n)$
6: Define $\hat{Y}_o \equiv \text{CNN}(X_n, \hat{\theta}_s, \hat{\theta}_o)$
7: $\theta_s^*, \theta_o^*, \theta_n^* \leftarrow \text{argmin}_{\hat{\theta}_s, \hat{\theta}_o, \hat{\theta}_n} \left(\mathcal{L}_{new}(\hat{Y}_n, Y_n) + \lambda_o \mathcal{L}_{old}(\hat{Y}_o, Y_o) + \mathcal{R}(\theta_s, \theta_o, \theta_n) \right)$

- $\mathcal{L}_{new}(\hat{Y}_n, Y_n)$: minimize the difference between the predicted values $\hat{Y}_n$ and the groundtruth Y_n. $\hat{Y}_n$ is the predicted value using the current parameters $\hat{\theta}_s$ and $\hat{\theta}_n$ (Line 5). In Li and Hoiem [2016], the multinomial logistic loss is used:

$$\mathcal{L}_{new}(\hat{Y}_n, Y_n) = -Y_n \cdot \log \hat{Y}_n \, .$$

- $\mathcal{L}_{old}(\hat{Y}_o, Y_o)$: minimize the difference between the predicted values $\hat{Y}_o$ and the recorded values Y_o (Line 2), where $\hat{Y}_o$ comes from the current parameters $\hat{\theta}_s$ and $\hat{\theta}_o$ (Line 6). Li and Hoiem [2016] used knowledge distillation loss [Hinton et al., 2015] to encourage the outputs of one network to approximate the outputs of another. The distillation loss is defined as modified cross-entropy loss:

$$\begin{aligned} \mathcal{L}_{old}(\hat{Y}_o, Y_o) &= -H(\hat{Y}_o', Y_o') \\ &= -\sum_{i=1}^{l} y_o'^{(i)} \log \hat{y}_o'^{(i)} \, , \end{aligned}$$

where l is the number of labels. $y_o'^{(i)}$ and $\hat{y}_o'^{(i)}$ are the modified probabilities defined as:

$$y_o'^{(i)} = \frac{(y_o^{(i)})^{1/T}}{\sum_j (y_o^{(j)})^{1/T}}, \quad \hat{y}_o'^{(i)} = \frac{(\hat{y}_o^{(i)})^{1/T}}{\sum_j (\hat{y}_o^{(j)})^{1/T}} \, .$$

T is set to 2 in Li and Hoiem [2016] to increase the weights of smaller logit values. In the objective function (Line 7), λ_o is used to balance the new task and the old/past tasks. Li and Hoiem [2016] tried various values for λ_o in their experiments.

- $\mathcal{R}(\theta_s, \theta_o, \theta_n)$: regularization term to avoid overfitting.

4.4 PROGRESSIVE NEURAL NETWORKS

Progressive neural networks were proposed by Rusu et al. [2016] to explicitly tackle catastrophic forgetting for the problem of LL. The idea is to keep a pool of pretrained models as knowledge, and use lateral connections between them to adapt to the new task. The model was originally proposed to tackle reinforcement learning, but the model architecture is general enough for other ML paradigms such as supervised learning. Assuming there are N existing/past tasks: $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N$, progressive neural networks maintain N neural networks (or N columns). When a new task $\mathcal{T}_{N+1}$ is created, a new neural network (or a new column) is created, and its lateral connections with all previous tasks are learned. The mathematical formulation is presented below.

In progressive neural networks, each task $\mathcal{T}_n$ is associated with a neural network, which is assumed to have L layers with hidden activations $h_i^{(n)}$ for the units at layer $i \leq L$. The set of parameters in the neural network for $\mathcal{T}_n$ is denoted by $\Theta^{(n)}$. When a new task $\mathcal{T}_{N+1}$ arrives, the parameters $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(N)}$ remain the same while each layer $h_i^{(N+1)}$ in the $\mathcal{T}_{N+1}$'s neural network takes inputs from $(i-1)$th layers of all previous tasks' neural networks, i.e.,

$$h_i^{(N+1)} = \max\left(0, W_i^{(N+1)} h_{i-1}^{(N+1)} + \sum_{n<N+1} U_i^{(n:N+1)} h_{i-1}^{(n)}\right) , \tag{4.1}$$

where $W_i^{(N+1)}$ denotes the weight matrix of layer i in neural network $N+1$. The lateral connections are learned via $U_i^{(n:N+1)}$ to indicate how strong the $(i-1)$th layer from task n influences the ith layer from task $N+1$. h_0 is the network input.

Unlike pretraining and fine-tuning, progressive neural networks do not assume any relationship between tasks, which makes it more practical for real-world applications. The lateral connections can be learned for related, orthogonal, or even adversarial tasks. To avoid catastrophic forgetting, settings of parameters $\Theta^{(n)}$ for existing tasks $\mathcal{T}_n$ where $n \leq N$ are "frozen" while the new parameter set $\Theta^{(N+1)}$ is learned and adapted for the new task $\mathcal{T}_{N+1}$. As a result, the performance on existing tasks does not degrade.

For the applications in reinforcement learning, each task's neural network is trained to learn a policy function for a particular Markov Decision Process (MDP). The policy function implies the probabilities over actions given states. Nonlinear lateral connections are learned through a single hidden perceptron layer, which reduces the number of parameters from the lateral connections to the same order as $|\Theta^{(1)}|$. More details can be found in Rusu et al. [2016].

With the flexibility of considering various task relationships, progressive neural networks come at a price: it can explode the numbers of parameters with an increasing number of tasks, since it needs to learn a new neural network for a new task and its lateral connections with all existing ones. Rusu et al. [2016] suggested pruning [LeCun et al., 1990] or online compression [Rusu et al., 2015] as potential solutions.

4.5 ELASTIC WEIGHT CONSOLIDATION

Kirkpatrick et al. [2017] proposed a model called *Elastic Weight Consolidation* (EWC) to mitigate catastrophic forgetting in neural networks. It was inspired by human brain in which synaptic consolidation enables continual learning by reducing the plasticity of synapses related to previous learned tasks. As mentioned in Section 4.1, plasticity is the main cause of catastrophic forgetting since the weights learned in the previous tasks can be easily modified given a new task. More precisely, plasticity of weights that are closely related to previous tasks is more prone to catastrophic forgetting than plasticity of weights that are loosely connected to previous tasks. This motivates [Kirkpatrick et al., 2017] to quantify the importance of weights in terms of their impact on previous tasks' performance, and selectively decrease the plasticity of those important weights to previous tasks.

Kirkpatrick et al. [2017] illustrated their idea using an example consisting of two tasks A and B where A is a previous task and B is the new task. The example only contains two tasks for easy understanding, but the EWC model works in an LL manner with tasks coming in a sequence. The parameters (weights and biases) for task A and B are represented by θ_A and θ_B. The sets of parameters that lead to low errors for task A and B are represented by Θ_A^* and Θ_B^*, respectively. Over-parametrization makes it possible to find a solution $\theta_B^* \in \Theta_B^*$ and $\theta_B^* \in \Theta_A^*$, i.e., the solution is learned toward task B while also maintaining low errors in task A. EWC achieves this goal by constraining the parameters to stay in A's low-error region. Figure 4.1 visualizes the example.

The Bayesian approach is used to measure the importance of parameters toward a task in EWC. In particular, the importance is modeled as the posterior distribution $p(\theta|\mathcal{D})$, the probability of parameter θ given a task's training data $\mathcal{D}$. Using Bayes' rule, the log value of the posterior probability is:

$$\log p(\theta|\mathcal{D}) = \log p(\mathcal{D}|\theta) + \log p(\theta) - \log p(\mathcal{D}) \ . \tag{4.2}$$

Assume that the data consists of two independent parts: $\mathcal{D}_A$ for task A and $\mathcal{D}_B$ for task B. Equation (4.2) can be written as:

$$\log p(\theta|\mathcal{D}) = \log p(\mathcal{D}_B|\theta) + \log p(\theta|\mathcal{D}_A) - \log p(\mathcal{D}_B) \ . \tag{4.3}$$

The left side in Equation (4.3) is still the posterior distribution given the *entire* dataset, while the right side only depends on the loss function for task B, i.e., $\log p(\mathcal{D}_B|\theta)$. All the information related to task A is embedded in the term $\log p(\theta|\mathcal{D}_A)$. EWC wants to extract information about weight importance from $\log p(\theta|\mathcal{D}_A)$. Unfortunately, $\log p(\theta|\mathcal{D}_A)$ is intractable. Thus, EWC approximates it as a Gaussian distribution with mean given by the parameters θ_A^* and a diagonal precision by the diagonal of the Fisher information matrix F. Thus, the new loss function in EWC is:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i (\theta_i - \theta_{A,i}^*)^2 \ , \tag{4.4}$$

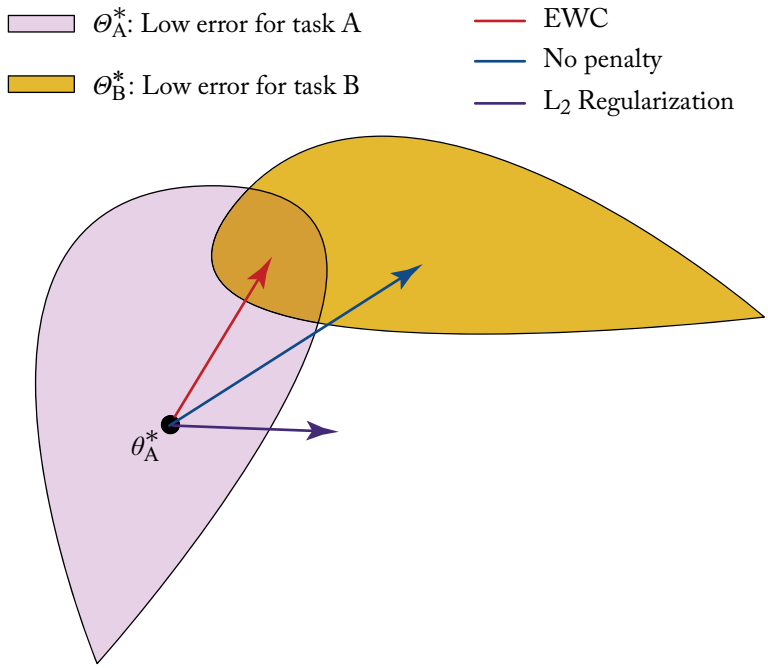

Figure 4.1: An example to illustrate EWC. Given task B, a regular neural network learns a point that yields a low error for task B but not task A (blue arrow). A L_2 regularization instead provides a suboptimal model to task B (purple arrow). EWC updates its parameters for task B while slowly updating the parameters important to task A to stay in A's low error region (red arrow).

where $\mathcal{L}_B(\theta)$ is the loss for task B only. λ controls how strong the constraint posed should not move too far away from task A's low error area. i denotes each index in the weight vector.

Recall that if θ has n dimensions, $\theta_1, \theta_2, \ldots, \theta_n$, the Fisher information matrix F is a $n \times n$ matrix with each entry being:

$$I(\theta)_{ij} = E_X \left[\left(\frac{\partial}{\partial \theta_i} \log p(\mathcal{D}|\theta) \right) \left(\frac{\partial}{\partial \theta_j} \log p(\mathcal{D}|\theta) \right) \middle| \theta \right] . \tag{4.5}$$

The diagonal entry is then:

$$F_i = I(\theta)_{ii} = E_X \left[\left(\frac{\partial}{\partial \theta_i} \log p(\mathcal{D}|\theta) \right)^2 \middle| \theta \right] . \tag{4.6}$$

When a task C comes, EWC updates Equation (4.4) with the penalty term enforcing the parameters θ to be close to $\theta^*_{A,B}$, where $\theta^*_{A,B}$ is the parameters learned for tasks A and B.

To evaluate EWC, Kirkpatrick et al. [2017] used the MNIST dataset [LeCun et al., 1998]. A new task is obtained by generating a random permutation and the input pixels of all images are shuffled according to the permutation. As a result, each task is unique with equal difficulty to the original MNIST problem. The results showed that EWC achieves superior performances to those models that suffer from catastrophic forgetting. For more details on the evaluation as well as EWC's application in reinforcement learning, please refer to the original paper by Kirkpatrick et al. [2017].

4.6 ICARL: INCREMENTAL CLASSIFIER AND REPRESENTATION LEARNING

Rebuffi et al. [2017] proposed a new model for *class-incremental learning*. Class-incremental learning requires the classification system to incrementally learn and classify new classes that it has never seen before. This is similar to *open-world-learning* (or *cumulative learning*) [Fei et al., 2016] introduced in Chapter 5 without the rejection capability of open-world learning. It assumes that examples of different classes can occur at different times, with which the system should maintain a satisfactory classification performance on each observed class. Rebuffi et al. [2017] also emphasized that computational resources should be bounded or slowly increased with more and more classes coming.

To meet these criteria, a new model called *iCaRL*, *incremental Classifier and Representation Learning*, was designed to simultaneously learn classifiers and feature representations in the class-incremental setting. At the high level, iCaRL maintains a set of exemplar examples for each observed class. For each class, an exemplar set is a subset of all examples of the class, aiming to carry the most representative information of the class. The classification of a new example is performed by choosing the class whose exemplars are the most similar to it. When a new class shows up, iCaRL creates an exemplar set for this new class while trimming the exemplar sets of the existing/previous classes.

Formally, at any time, iCaRL learns a stream of classes in the class-incremental learning setting with their training example sets, X^s, X^{s+1}, ..., X^t, where X^y is a set of examples of class y. y can either be an observed/past class or a new class. To avoid memory overflow, iCaRL holds a fixed number (K) of exemplars in total. With C classes, the exemplar sets are represented by $\mathcal{P} = \{P_1, \ldots, P_C\}$ where each class's exemplar set P_i maintains K/C exemplars. In Rebuffi et al. [2017], both original examples and exemplars are images, but the proposed method is general enough for non-image datasets.

4.6.1 INCREMENTAL TRAINING

Algorithm 4.2 presents the incremental training algorithm in iCaRL with new training example sets X^s, ..., X^t of classes s, ..., t arriving in a stream. Line 1 updates the model parameters Θ with the new training examples (defined in Algorithm 4.3). Line 2 computes the number of

Algorithm 4.2 iCaRL IncrementalTraining

Input: new training examples $X^s, \ldots, X^t$ of new classes $s, \ldots, t$, current model parameters Θ, current exemplar sets $\mathcal{P} = \{P_1, \ldots, P_{s-1}\}$, memory size K.
Output: updated model parameters Θ, updated exemplar sets $\mathcal{P}$.

1: $\Theta \leftarrow$ UpdateReresentation$(X^s, \ldots, X^t; \mathcal{P}, \Theta)$
2: $m \leftarrow K/t$
3: **for** $y = 1$ **to** $s - 1$ **do**
4: $\quad P_y \leftarrow P_y[1 : m]$
5: **end for**
6: **for** $y = s$ **to** t **do**
7: $\quad P_y \leftarrow$ ConstructExemplarSet(X^y, m, Θ)
8: **end for**
9: $\mathcal{P} \leftarrow \{P_1, \ldots, P_t\}$

exemplars per class. For each existing class, we reduce the number of exemplars per class to m. Since the exemplars are created in the order of importance (see Algorithm 4.4), we just keep the first m exemplars for each class (Line 3–5). Line 6–8 construct the exemplar set for each new class (see Algorithm 4.4).

4.6.2 UPDATING REPRESENTATION

Algorithm 4.3 details the steps for updating the feature representation. Two datasets are created (Lines 1 and 2): one with all existing exemplar examples, and the other with new examples of the new classes. Note that the exemplar examples have the original feature space, not the learned representation. Lines 3–5 store the prediction output of each exemplar example with the current model. Learning in Rebuffi et al. [2017] used a convolutional neural network (CNN) [LeCun et al., 1998], interpreted as a trainable feature extractor: $\varphi : \mathcal{X} \rightarrow \mathbb{R}^d$. A single classification layer is added with as many sigmoid output nodes as the number of classes observed so far. The output score for class $y \in \{1, \ldots, t\}$ is formulated as follows:

$$g_y(x) = \frac{1}{1 + exp(-a_y(x))} \qquad \text{with } a_y(x) = w_y^{\mathrm{T}} \varphi(x) \ . \tag{4.7}$$

Note that the network is just utilized for representation learning, not for the actual classification. The actual classification is covered in Section 4.6.4. The last step in Algorithm 4.3 runs Backpropagation with the loss function that (1) minimizes the loss on the new examples of new classes D^{new} (*classification loss*), and (2) reproduces the scores stored using previous networks (*distillation loss* [Hinton et al., 2015]). The hope is that the neural network will be updated with new examples of the new classes, while not forgetting the existing classes.

Algorithm 4.3 iCaRL UpdateRepresentation

Input: new training examples $X^s, \ldots, X^t$ of new classes $s, \ldots, t$, current model parameters Θ, current exemplar sets $\mathcal{P} = \{P_1, \ldots, P_{s-1}\}$, memory size K.
Output: updated model parameters Θ.

1: $\mathcal{D}^{exemplar} \leftarrow \bigcup_{y=1,\ldots,s-1} \{(x, y) : x \in P_y\}$
2: $\mathcal{D}^{new} \leftarrow \bigcup_{y=s,\ldots,t} \{(x, y) : x \in X^y\}$
3: **for** $y = 1$ **to** $s - 1$ **do**
4: $\quad q_i^y \leftarrow g_y(x_i)$ for all $(x_i, \cdot) \in \mathcal{D}^{exemplar}$
5: **end for**
6: $\mathcal{D}^{train} \leftarrow \mathcal{D}^{exemplar} \cup \mathcal{D}^{new}$
7: Run network training (e.g., Backpropagation) with loss function that contains *classification* and *distillation* terms:

$$\mathcal{L}(\Theta) = -\sum_{(x_i,y_i)\in\mathcal{D}^{train}} [\sum_{y=s}^{t} \delta_{y=y_i} \log g_y(x_i) + \delta_{y\neq y_i} \log(1 - g_y(x_i)) + \sum_{y=1}^{s-1} q_i^y \log g_y(x_i) + (1 - q_i^y) \log(1 - g_y(x_i))]$$

4.6.3 CONSTRUCTING EXEMPLAR SETS FOR NEW CLASSES

When examples of a new class t show up, iCaRL balances the number of exemplars in each class, i.e., reducing the number of exemplars for each existing class and creating the exemplar set for the new class. If K exemplars are allowed in total due to the memory limitation, each class receives $m = K/t$ exemplar quota. For each existing class, the first m exemplars are kept (Lines 3–5 in Algorithm 4.2). For the new class t, Algorithm 4.4 chooses m exemplars for it. Here is the intuition of how the selection of exemplars works: the average feature vector over all exemplars should be close to the average feature vector over all examples of the class. As such, the general property of all examples in a class does not diminish much when most of them are removed, i.e., only exemplars are retained. Also, to make sure that exemplars can be easily trimmed, the exemplars are stored in the order that the most important ones are stored first, thus making the list a priority list.

In Algorithm 4.4, the average feature vector μ of all training examples of class t is computed (Line 1). Then m exemplars are selected in the order that by picking each exemplar p_k, the average feature vector is the closest to μ compared to adding any other non-exemplar example (Lines 2–4). Consequently, the resulting exemplar set $P \leftarrow (p_1, \ldots, p_m)$ should well approximate the class mean vector. Note that all non-exemplar examples are dropped after class t training. So having an ordered list of exemplars according to importance is a key to LL since it is easy to reduce its size with future new classes added while retaining the most essential past information.

Algorithm 4.4 iCaRL ConstructExemplarSet

Input: examples $X = \{x_1, \ldots, x_n\}$ of class t, the target number of exemplars m, current feature function $\varphi : \mathcal{X} \rightarrow \mathbb{R}^d$.
Output: exemplar set P for class y.

$\mu \leftarrow \frac{1}{n} \sum_{x \in X} \varphi(x)$
for $k = 1$ **to** m **do**
  $p_k \leftarrow \text{argmin}_{x \in X \text{and} x \notin \{p_1, \ldots, p_{k-1}\}} \left|\left| \mu - \frac{1}{k} [\varphi(x) + \sum_{j=1}^{k-1} \varphi(p_j)] \right|\right|$
end for
$P \leftarrow (p_1, \ldots, p_m)$

4.6.4 PERFORMING CLASSIFICATION IN ICARL

With all the training algorithms introduced above, the classification is performed with the sets of exemplars $\mathcal{P} = \{P_1, \ldots, P_t\}$. The idea is straightforward: given a test example x, we pick the class y^* whose exemplar set's average feature vector is the closet to x as x's class label (see Algorithm 4.5).

Algorithm 4.5 iCaRL Classify in iCaRL

Input: a test example x to be classified, sets of exemplars $\mathcal{P} = \{P_1, \ldots, P_t\}$, current feature function $\varphi : \mathcal{X} \rightarrow \mathbb{R}^d$.
Output: predicted class label y^* of x.

for $y = 1$ **to** t **do**
  $\mu_y \leftarrow \frac{1}{|P_y|} \sum_{p \in P_y} \varphi(p)$
end for
$y^* \leftarrow \underset{y=1,\ldots,t}{\text{argmin}} ||\varphi(x) - \mu_y||$

4.7 EXPERT GATE

Aljundi et al. [2016] proposed a *Network of Experts* where each expert is a model trained given a specific task. Since an expert is trained on one task only, it is good at this particular task, but not others. Thus, in the LL context, a network of experts are needed to handle a sequence of tasks.

One compelling point that Aljundi et al. [2016] emphasizes is the importance of *memory efficiency*, especially in the era of big data. As we know, GPUs are widely used for training deep learning models due to their rapid processing capability. However, GPUs have limited memory compared to CPUs. As deep learning models are becoming more and more complex, GPUs can

only load a small number of models at a time. With a large number of tasks, as in LL, it requires the system to know what model or models to load when making a prediction on a test example.

With this need in mind, Aljundi et al. [2016] proposed an *Expert Gate* algorithm to determine the relevance of tasks, and only load the most relevant tasks in memory during inference. Denoting the existing tasks as $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_N$, an undercomplete autoencoder model A_k and an expert model E_k are constructed for each existing task $\mathcal{T}_k$ where $k \in \{1, \ldots, N\}$. When a new task $\mathcal{T}_{N+1}$ and its training data $\mathcal{D}_{N+1}$ arrive, $\mathcal{D}_{N+1}$ will be evaluated against each autoencoder A_k to find the most relevant tasks. The expert models of these most relevant tasks are used for fine-tuning or learning-without-forgetting (LwF) (Section 4.3) to build the expert model E_{N+1}. At the same time, A_{N+1} is learned from $\mathcal{D}_{N+1}$. When making a prediction on a test example x_t, the expert models whose corresponding autoencoders best describe x_t are loaded in memory and used to make the prediction.

4.7.1 AUTOENCODER GATE

An autoencoder [Bourlard and Kamp, 1988] model is a neural network that learns to recover input in the output layer in an unsupervised manner. There are encoder and decoder in the model. The encoder $f = h(x)$ projects the input x to an embedded space $h(x)$ while the decoder $r = g(h(x))$ maps the embedded space to the original input space. There are two types of autoencoder models: undercomplete autoencoder and overcomplete autoencoder. Undercomplete autoencoder learns a lower-dimensional representation and overcomplete autoencoder learns a higher-dimensional representation with regularization. An example of undercomplete autoencoder is illustrated in Figure 4.2.

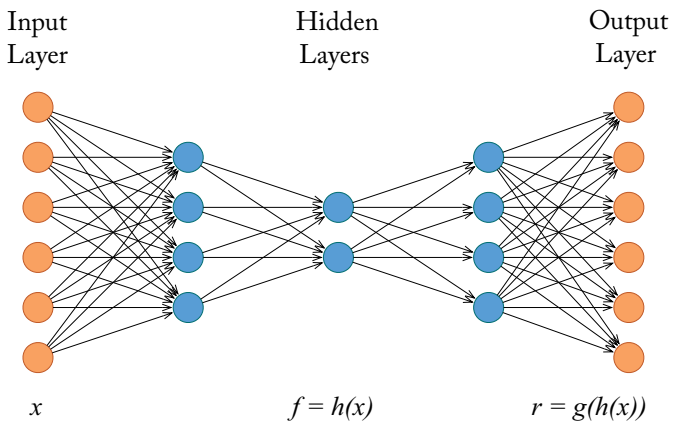

Figure 4.2: An example of undercomplete autoencoder model.

The intuition for using autoencoder in Expert Gate is that, as an unsupervised approach, an undercomplete autoencoder can learn a lower-dimensional feature representation that best

describes the data in a compact way. The autoencoder of one task should perform well at reconstructing the data of that task, i.e., one autoencoder model is a decent representation of one task. If two autoencoder models from two tasks are close to each other, the tasks are likely to be similar too.

The autoencoder used in Aljundi et al. [2016] is simple: it has one ReLU layer [Zeiler et al., 2013] between the encoding and decoding layers. ReLU activation units are fast and easy to optimize, which also introduce sparsity to avoid over-fitting.

4.7.2 MEASURING TASK RELATEDNESS FOR TRAINING

Given a new task $\mathcal{T}_{N+1}$ with its training data $\mathcal{D}_{N+1}$, Expert Gate first learns an autoencoder A_{N+1} from $\mathcal{D}_{N+1}$. To facilitate training expert model E_{N+1}, it finds the most related existing task and use its expert model. Specifically, the reconstruction error of $\mathcal{D}$ using an autoencoder A_k is defined as:

$$Er_k = \frac{\sum_{x \in \mathcal{D}} er_x^k}{|\mathcal{D}|} \quad , \tag{4.8}$$

where er_x^k is the reconstruction error of applying x to the autoencoder A_k. Since the data of existing tasks are discarded, only $\mathcal{D}_{N+1}$ can be used to evaluate the relatedness. Given an existing task $\mathcal{T}_k$, $\mathcal{D}_{N+1}$ is used to compute two reconstruction errors: Er_{N+1} of autoencoder A_{N+1} and E_k of autoencoder A_k. The task relatedness is thus defined as:

$$Relatedness(\mathcal{T}_{N+1}, \mathcal{T}_k) = 1 - \frac{Er_{N+1} - Er_k}{Er_k} \quad . \tag{4.9}$$

Note that this relatedness definition is asymmetric. After the most related task is chosen, depending on how related it is to the new task, fine-tuning (see Section 4.1) or learning-without-forgetting (LwF) (Section 4.3) is employed. If two tasks are sufficiently related, LwF is applied; otherwise, fine-tuning is used. In LwF, a shared model is used for all tasks while each task has its own classification layer. A new task introduces a new classification layer. The model is fine-tuned to the new task's data while trying to preserve the previous tasks' predictions on the new data.

4.7.3 SELECTING THE MOST RELEVANT EXPERT FOR TESTING

If a test example x_t yields a very small reconstruction error when going through an autoencoder (say A_k), x_t should be similar to the data that was used to train A_k. The specialized model (expert) E_k should hence be utilized to make predictions on x_t. The probability p_k of x_t being relevant to an expert E_k is defined as:

$$p_k = \frac{exp(-er_{x_t}^k / \tau)}{\sum_j exp(-er_{x_t}^j / \tau)} \quad , \tag{4.10}$$

where $er_{x_t}^k$ is the reconstruction error of applying x_t to the autoencoder A_k. τ is the temperature whose value is 2, leading to soft probability values. Aljundi et al. [2016] picked the expert $E_{k'}$

to make the prediction on x_t whose $p_{k'}$ is the maximum among all existing tasks. The approach can also accommodate loading multiple experts by simply selecting experts whose relevant score is higher than a threshold.

4.7.4 ENCODER-BASED LIFELONG LEARNING

Finally, we note that Rannen Ep Triki et al. [2017] also used the idea of autoencoder to extend LwF (Section 4.3). Rannen Ep Triki et al. [2017] argued that LwF has an inferior loss function definition when the new task data distribution is quite different from those of previous tasks. To address this issue, an autoencoder based method is proposed to preserve only the features that are the most important for previous tasks while allowing other features to adapt more quickly to new tasks. This is achieved by learning a lower-dimensional manifold via autoencoder, and constraining the distance between the reconstructions. Note that this is similar to EWC (Section 4.5) in the sense that EWC tries to maintain the most important weights while Rannen Ep Triki et al. [2017] aims to conserve features. See Rannen Ep Triki et al. [2017] for more details.

4.8 CONTINUAL LEARNING WITH GENERATIVE REPLAY

Shin et al. [2017] proposed a continual learning method using replayed examples from a generative model without referring to the actual data of past tasks. It is inspired by the suggestion that the hippocampus is better paralleled with a generative model than a replay buffer [Ramirez et al., 2013, Stickgold and Walker, 2007]. As mentioned in Section 4.2, this represents a stream of lifelong learning systems that use dual-memory for knowledge consolidation. We pick the work of Shin et al. [2017] to give a flavor of such models. In the deep generative replay framework proposed by Shin et al. [2017], a generative model is maintained to feed pseudo-data as knowledge of past tasks to the system. To train such a generative model, the generative adversarial networks (GANs) [Goodfellow et al., 2014] framework is used. Given a sequence of tasks, a *scholar* model, containing a generator and a solver, is learned and retained. Such a scholar model holds the knowledge representing the previous tasks, and thus prevents the system from forgetting previous tasks.

4.8.1 GENERATIVE ADVERSARIAL NETWORKS

The Generative Adversarial Networks (GANs) framework is not only used in Shin et al. [2017], but also widely adopted in the deep learning community (e.g., Radford et al. [2015]). In this subsection, we give an overview of GANs based on Goodfellow [2016].

In GANs, there are two players: a *generator* and a *discriminator*. On the one hand, the generator creates samples that mimic training data, i.e., drawing samples from the similar (ideally same) distribution as the training data. On the other hand, the discriminator classifies the samples to tell whether they are real (from real training data) or fake (from samples created by the generator). The problem that discriminator faces is a typical binary classification prob-

lem. Following the example given in Goodfellow [2016], a generator is like a counterfeiter who tries to make fake money. A discriminator is like a police who wants to allow legitimate money and catch counterfeit money. To win the game, the counterfeiter (generator) must learn how to make money that looks identical to genuine money while the police (discriminator) learns how to distinguish authenticity without mistakes.

Formally, GANs are a structured probabilistic model with latent variables $\boldsymbol{z}$ and observed variables $\boldsymbol{x}$. The discriminator has a function D that takes $\boldsymbol{x}$ as input. The function for the generator is defined as G whose input is $\boldsymbol{z}$. Both functions are differentiable with respect to their inputs and parameters. The cost function for the discriminator is:

$$J = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \ . \tag{4.11}$$

By treating the two-player game as a *zero-sum game* (or *minimax* game), the solution involves minimization in an outer loop and maximization in an inner loop, yielding the objective function for discriminator D and generator G as:

$$\begin{aligned}\mathcal{L}(D, G) &= \min_G \max_D V(D, G) \\ &= \min_G \max_D -J \\ &= \min_G \max_D \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \ .\end{aligned} \tag{4.12}$$

4.8.2 GENERATIVE REPLAY

In Shin et al. [2017], a *scholar* model H is learned and maintained in an LL manner. The scholar model contains a generator G and a solver S with parameters θ. The solver here is like the discriminator in Section 4.8.1. Denoting the previous N tasks as $\boldsymbol{\mathcal{T}}_N = (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N)$, and the scholar model for previous N task as $H_N = \langle G_N, S_N \rangle$, the system aims to learn a new scholar model $H_{N+1} = \langle G_{N+1}, S_{N+1} \rangle$ given the new task T_{N+1}'s training data $\mathcal{D}_{N+1}$.

To obtain $H_{N+1} = \langle G_{N+1}, S_{N+1} \rangle$ given the training data $\mathcal{D}_{N+1} = (\boldsymbol{x}, \boldsymbol{y})$, there are two steps.

1. G_{N+1} is updated with the new task input $\boldsymbol{x}$ and replayed inputs $\boldsymbol{x}'$ created from G_N. Real and replayed samples are mixed at a ratio that depends on the importance of the new task compared to previous ones. Recall that this step is known as intrinsic replay or pseudo-rehearsal [Robins, 1995] in which new data and replayed samples of old data are mixed to prevent catastrophic forgetting.

2. S_{N+1} is trained to couple the inputs and targets drawn from the same mix of real and replayed data, with the loss function:

$$\begin{aligned}\mathcal{L}_{train}(\theta_{N+1}) = r\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}_{N+1}}[L(S(\boldsymbol{x}; \theta_{N+1}), \boldsymbol{y})] \\ + (1 - r)\mathbb{E}_{\boldsymbol{x}'\sim G_N}[L(S(\boldsymbol{x}'; \theta_{N+1}), S(\boldsymbol{x}'; \theta_N))] \ ,\end{aligned} \tag{4.13}$$

where θ_N denotes the parameters for the solver S_N, and r denotes the ratio of mixing real data. If S_N is tested on the previous tasks, the test loss function becomes:

$$\begin{aligned}\mathcal{L}_{test}(\theta_{N+1}) = r\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}_{N+1}}[L(S(\boldsymbol{x};\theta_{N+1}),\boldsymbol{y})] \\ + (1-r)\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}_{past}}[L(S(\boldsymbol{x};\theta_{N+1}),\boldsymbol{y})]\ ,\end{aligned} \tag{4.14}$$

where $\mathcal{D}_{past}$ is the cumulative distribution of the data from the past tasks.

The proposed framework is independent of any specific generative model or solver. The choice for the deep generative model can be a variational autoencoder [Kingma and Welling, 2013] or a GAN [Goodfellow et al., 2014].

4.9 EVALUATING CATASTROPHIC FORGETTING

There are two main papers [Goodfellow et al., 2013a, Kemker et al., 2018] in the literature that evaluate ideas aimed at addressing catastrophic forgetting in neural networks.

Goodfellow et al. [2013a] evaluated some traditional approaches that attempt to reduce catastrophic forgetting. They evaluated dropout training [Hinton et al., 2012] as well as various activation functions including:

- logistic sigmoid,
- rectified linear [Jarrett et al., 2009],
- hard local winner take all (LWTA) [Srivastava et al., 2013], and
- Maxout [Goodfellow et al., 2013b].

They also used random hyperparameter search [Bergstra and Bengio, 2012] to automatically select hyperparameters. In terms of experiments, only pairs of tasks were considered in Goodfellow et al. [2013a] with one being the "old task" and the other being the "new task." The tasks were MNIST classification [LeCun et al., 1998] and sentiment classification on Amazon reviews [Blitzer et al., 2007]. Their experiments showed that dropout training is mostly beneficial to prevent forgetting. They also found that the choice of activation function matters less than the choice of training algorithm.

Kemker et al. [2018] evaluated several more recent continual learning algorithms using larger datasets. These algorithms include the following.

- Elastic weight consolidation (EWC) [Kirkpatrick et al., 2017]: it reduces plasticity of important weights with respect to previous tasks when adapting to a new task (see Section 4.5).
- PathNet [Fernando et al., 2017]: it creates an independent output layer for each task to preserve previous tasks. It also finds the optimal path to be trained when learning a particular task, which is like a dropout network.

- GeppNet [Gepperth and Karaoguz, 2016]: it reserves a sample set of training data of previous tasks, which is replayed to serve as a short-term memory when training on a new task.
- Fixed expansion layer (FEL) [Coop et al., 2013]: it uses sparsity in representation to mitigate catastrophic forgetting.

They proposed three benchmark experiments for measuring catastrophic forgetting.

1. **Data Permutation Experiment**: The elements in the feature vector are randomly permutated. In the same task, the permutation order is the same while different tasks have distinct permutation orders. This is similar to the experiment setup in Kirkpatrick et al. [2017].
2. **Incremental Class Learning**: After learning the base task set, each new task contains only a single class to be incrementally learned.
3. **Multi-Modal Learning**: The tasks contain different datasets, e.g., learn image classification and then audio classification.

Three datasets were used in the experiments: MNIST [LeCun et al., 1998], CUB-200 [Welinder et al., 2010], and AudioSet [Gemmeke et al., 2017]. Kemker et al. [2018] evaluated the accuracy on the new task as well as the old tasks in the LL setting, i.e., tasks arriving in a sequence. They found that PathNet performs the best in data permutation, GreppNet obtains the best accuracy in incremental class learning, and EWC has the best results in multi-modal learning.

4.10 SUMMARY AND EVALUATION DATASETS

This chapter reviewed the problem of catastrophic forgetting and existing continual learning algorithms aimed at dealing with it. Most existing works fall into some variations of regularization or increasing/allocating extra parameters for new tasks. They are shown to be effective in some simplified LL settings. Considering the huge success of deep learning in recent years, continual/lifelong deep learning continues to be one of the most promising channels to reach true intelligence with embedded LL. Nonetheless, catastrophic forgetting remains a long-standing challenge. We look forward to the day when a robot can learn to perform all kinds of tasks and solve all kinds of problems continually and seamlessly without human intervention and without interfering each other.

To reach this ideal, there are many obstacles and gaps. We believe that one major gap is how to seamlessly discover, integrate, organize, and solve problems or tasks of different similarities at the different levels of detail in a single network or even multiple networks just like our human brains, with minimum interference of each other. For example, some tasks are dissimilar at the detailed action level but may be similar at a higher or more abstract level. How to automatically recognize and leverage the similarities and differences in order to learn quickly

and better in an incremental and lifelong manner without the need of a large amount of training data is a very challenging and interesting research problem.

Another gap is the lack of research in designing systems that can truly embrace real-life problems with memories. This is particularly relevant to DNNs due to catastrophic forgetting. One idea is to encourage the system to take snapshots of its status and parameters, and keep validating itself against a gold dataset. It is not practical to retain all the training data. But to prevent the system from moving to some extreme parameter point in the space, it is useful to keep a small sampled set of training data that can cover most of the patterns/classes seen before.

In short, catastrophic forgetting is a key challenge for DNNs to enable LL. We hope this chapter can shed some light in the area and attract more attention to address this challenge.

Regarding evaluation datasets, image data are among the most commonly used datasets for evaluating continual learning due to their wide availability. Some of the common ones are as follows.

- **MNIST** [LeCun et al., 1998][1] is perhaps the most commonly used dataset (used in more than half of the works introduced in this chapter). It consists of labeled examples of handwritten digits. There are 10 digit classes. One way to produce datasets for multiple tasks is to create the representations of the data by randomly permuting the elements of input feature vectors [Goodfellow et al., 2013a, Kemker et al., 2018, Kirkpatrick et al., 2017]. This paradigm ensures that the tasks are overlapping and have equal complexity.

- **CUB-200** (Caltech-UCSD Birds 200) [Welinder et al., 2010][2]) is another popular dataset for LL evaluation. It is an image dataset with photos of 200 bird species. It has been used in Aljundi et al. [2016, 2017], Kemker et al. [2018], Li and Hoiem [2016], Rannen Ep Triki et al. [2017], and Rosenfeld and Tsotsos [2017].

- **CIFAR-10** and **CIFAR-100** [Krizhevsky and Hinton, 2009][3] are also widely used. They contain images of 10 classes and 100 classes, respectively. They are used in Fernando et al. [2017], Jung et al. [2016], Lopez-Paz et al. [2017], Rebuffi et al. [2017], Venkatesan et al. [2017], Zenke et al. [2017], and Rosenfeld and Tsotsos [2017].

- **SVHN** (Google Street View House Numbers) [Netzer et al., 2011][4] is similar to MNIST, but contains an order of magnitude more labeled data. These images are from real-world problems and are harder to solve. It also has 10 digit classes. It is used in Aljundi et al. [2016, 2017], Fernando et al. [2017], Jung et al. [2016], Rosenfeld and Tsotsos [2017], Shin et al. [2017], Venkatesan et al. [2017], and Seff et al. [2017].

[1] http://yann.lecun.com/exdb/mnist/
[2] http://www.vision.caltech.edu/visipedia/CUB-200.html
[3] https://www.cs.toronto.edu/~kriz/cifar.html
[4] http://ufldl.stanford.edu/housenumbers/

Other image datasets include **Caltech-256** [Griffin et al., 2007],[5] **GTSR** [Stallkamp et al., 2012],[6] **Human Sketch dataset** [Eitz et al., 2012],[7] **Daimler** (DPed) [Munder and Gavrila, 2006],[8] **MIT Scenes** [Quattoni and Torralba, 2009],[9] **Flower** [Nilsback and Zisserman, 2008],[10] **FGVC-Aircraft** [Maji et al., 2013],[11] **ImageNet ILSVRC2012** [Russakovsky et al., 2015],[12] and **Letters** (Chars74K) [de Campos et al., 2009].[13]

More recently, Lomonaco and Maltoni [2017] proposed a dataset called **CORe50**.[14] It contains 50 objects that were collected in 11 distinct sessions (8 indoor and 3 outdoor) differing in background and lighting. The dataset is specifically designed for continual object recognition. Unlike many popular datasets such as **MNIST** and **SVHN**, **CORe50**'s multiple views of the same object from different sessions enable richer and more practical LL. Using **CORe50**, Lomonaco and Maltoni [2017] considered evaluation settings where the new data can contain (1) new patterns of the existing classes, (2) new classes, and (3) new patterns and new classes. Such real-life evaluation scenarios are very useful for carrying the LL research forward. Parisi et al. [2018b] used **CORe50** to perform an evaluation of their own approach as well as some other approaches, e.g., LwF [Li and Hoiem, 2016], EWC [Kirkpatrick et al., 2017], and iCaRL [Rebuffi et al., 2017].

Apart from image datasets, some other types of data are also used. **AudioSet** [Gemmeke et al., 2017][15] is a large-scale collection of human-labeled 10-sec sound clips sampled from YouTube videos. It is used in Kemker et al. [2018].

In continual learning on reinforcement learning, different environments were used for evaluation. **Atari games** [Mnih et al., 2013] are among the most popular ones which are used in Kirkpatrick et al. [2017], Rusu et al. [2016], and Lipton et al. [2016]. Some other environments include **Adventure Seeker** [Lipton et al., 2016], **CartPole-v0** in OpenAI gym [Brockman et al., 2016], and **Treasure World** [Mankowitz et al., 2018].

[5] http://ufldl.stanford.edu/housenumbers/
[6] http://benchmark.ini.rub.de/
[7] http://cybertron.cg.tu-berlin.de/eitz/projects/classifysketch/
[8] http://www.gavrila.net/Datasets/Daimler_Pedestrian_Benchmark_D/daimler_pedestrian_benchmark_d.html
[9] http://web.mit.edu/torralba/www/indoor.html
[10] http://www.robots.ox.ac.uk/~vgg/data/flowers/
[11] http://www.robots.ox.ac.uk/~vgg/data/fgvc-aircraft/
[12] http://www.image-net.org/challenges/LSVRC/
[13] http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/
[14] https://vlomonaco.github.io/core50/benchmarks.html
[15] https://research.google.com/audioset/dataset/index.html