The Why and How of Architectural Methods for Continual Learning

Léo Gagnon

Abstract

Current deep learning systems have been developed in the context of solving single task with access to carefully shuffled and homogeneous data. For this reason, when these learning algorithm are faced with sequential and non-stationary data steam–more like we humans do–they fail catastrophically. The main failure mode is called catastrophic interference and manifests itself as the constant overriding of old knowledge by new one. In this paper, we take a first-principles approach to explaining how this problem arises and what are the different ways to solve it : replay, regularization and architectural approaches. In particular, we focus our attention on the last class of method, as it is in some sort the continuation of the first two and because (as is often the case with deep learning) the current state of the field can feel like a bag of tricks without real direction. We focus our survey on how the different approaches work mechanistically and aim to be as self-contained as possible, but we do not include experimental comparison.

1. Introduction

As learning agents, information about the world comes to us sequentially and it can change drastically with time : at one moment we can be learning guitar and the next reading a book. Because of that, we need to be able to store knowledge and continuously build upon it, without forgetting. We call this ability that humans possess *continual learning*. For example, if you do 1 hour of guitar every day you will eventually become a good guitarist because the skill will accumulate and won't be forgotten in between sessions.

Unlike humans however, current artificial neural networks (ANN) systems are not designed to deal with data in that form. Instead it is normally assumed that the input data consists of many independent samples from a fictitious distribution $\mathcal{P}$ (the world). This is important because the learning algorithm on which is built all of modern deep learning–stochastic gradient descent (SGD)–works by aggressively rewiring the whole network to perform well on the newly seen data. When every new sample is *independent and identically distributed* (i.i.d.), this balances out to good performance on any sample, but when data is temporally correlated and non-homogeneous the network suffers from what is called "catastrophic interference" : parameters necessary for some kind of observation get overridden (i.e. forgotten) if said kind of observation doesn't happen for some time.

This problem, of learning from non-homogeneous sequential data with deep learning, is formalized and studied by the field of Continual Learning (CL). Concretely, a single neural network model is asked to solve a sequence of normal supervised learning tasks (i.i.d. data) one by one without forgetting the previous one. As the knowledge of a model is contained in its parameters, parameters should be reused, consolidated but not overridden when a new task is learned.

Different approaches to this problem are usually classified in the three categories : replay, regularization and architectural methods (Delange et al., 2021). First, replay-based methods try to artificially mix the sequential data and make it more homogeneous in order to make the usual optimization technique work. While this gives good results it scales poorly and avoid the problem more than it solves it. Then, regularization methods take a principled Bayesian approach to continual learning and constrain the optimization of every task with a prior based on previous tasks solutions. While this is theoretically sound, in practice it leads to clunky models that quickly lose capacity as the prior forces the later models to reuse entangled and sub-optimal features learned for previous tasks. To remedy to this problem, architectural approaches design various mechanisms which control how the representations of different tasks interact so as to maintain a balance between stability of old knowledge and plasticity to new one. This work focuses its attention on this line of work.

Our main contributions are the following :

- A comprehensive description of the problem of catastrophic interference and the ways solve it. The different approaches are introduces from first principles and their relationship made clear.
- An exhaustive (to the best of our knowledge) survey of the architectural-based continual learning approaches. We explain intuitively the important component of every paper and how they related, in contrast to one-line

description given by other continual learning surveys.

To the best of our knowledge, this is the first work to introduce continual learning approaches in such a comprehensible and intuitive form. Also, no other survey give such a broad survey of architectural methods.

2. Continual learning

Continual learning is defined vaguely by the sequential nature of the data stream. However, in order to make the problem more tractable we have to make things more concrete.

2.1. Formal definition

The continual learning setup is the following :

- The training data consists of T supervised learning tasks $\mathcal{D}_t = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N_t}$ where $\boldsymbol{x}_i \in \mathcal{X}_t$ and $y_i \in \mathcal{Y}_t$. We note $\mathcal{D} = \cup_{t=1}^{T} \mathcal{D}_t$.
- The training process goes through each task **only once** and at every task has access to only $\mathcal{D}_t$ to perform classic batch i.i.d. supervised learning.
- At every step, we have to learn a predictive function $\mathcal{X}_t \to \mathcal{Y}_t : f_t(\boldsymbol{x}; \theta_t) = y$ where $\theta_t \subseteq \theta_{t+1}$. Parameters are accumulated and hopefully reused.
- While learning task t, the goal is to minimize the empirical risk over all tasks seen so far

$$\sum_{i=1}^{t} \mathbb{E}_{(\boldsymbol{x},y)\in D_t}[l(f_i(\boldsymbol{x}; \theta_t), y)] \tag{1}$$

We can further define two more specific scenario considered in practice : task-incremental learning (Task-IL) and class-incremental learning (Class-IL). In Task-IL, the model is given a task identifier during inference while in Class-ID it needs to infer it. Also, because the tasks don't have the same number of labels or simply to simplify the problem, it is common for models to have a different output layer for every task.

To make this more concrete, we introduce tasks often used as benchmarks.

2.2. Common benchmarks

2.2.1. Permuted MNIST, (Goodfellow et al., 2013)

Here, every task is the result of applying a random permutation to every image in the MNIST dataset. Thus every tasks are the same difficulty (the data manifold is the same, the axis are just permuted) but they are also independent in a statistical sense. The output layer can be shared. This can be used in the Task-IL or Class-IL setting.

2.2.2. Rotated MNIST, (Larochelle et al., 2007)

Similar to Permuted MNIST but instead of a random permutation, a random rotation is applied. This makes the different tasks share more features and in a more meaningful way. The output layer can also be shared.

2.2.3. Split CIFAR100, (Zenke et al., 2017)

Here, the CIFAR dataset is split into T classification tasks each with $100/T$ classes. Since the output doesn't mean the same thing from task to task, one output layer per task is used. This can be used in the Task-IL or Class-IL setting.

2.2.4. Sequential recognition, (Delange et al., 2021)

Here, the different tasks can be any vision dataset. The number and classes and samples per tasks can therefore vary. One output layer per task is also used.

2.3. Important notions

When discussing continual learning algorithms, the following concepts are often used :

- *Backward transfer* (BWT), which is the influence that learning a task t has on the performance on a previous task t'. Backward transfer can be *positive* when performance on task t' increases and *negative* when it decreases.
- *Forward transfer* (FWT), which is the influence that learning a task t has on the performance on a future task t'. It often manifests itself as the ability to learn a future task quicker, with less data.
- *Scaling*, which is the rate at which the models grow, both in terms of compute and of memory. A desirable property, proposed by (MNTDP), is that the models *scale sublinearly* with the number of task.
- *The Stability/Plasticity dilema*, which is the fact that a continual learning system must be sensible (plastic), but not disrupted by, new information. Striking this balance is hard.

3. Interference : problem and solutions

Let us illustrate the big problem that arises when one tries to naïvely do continual learning with a single model and gradient-descent. Suppose you train a CNN on the CIFAR100 dataset. That would essentially amount to repeating

the following steps over and over until convergence (we omit batches for simplicity since they are not necessary to the discussion):

1. pick an image $\boldsymbol{x}$ randomly from the dataset
2. forward pass: compute the prediction $y = f(\boldsymbol{x}; \theta)$
3. backward pass: nudge every parameter in the direction that gets the prediction closer to the ground truth, i.e. the direction of the negative gradient of the loss $-\nabla_\theta l(y, \hat{y})$

At every step, a random image $\boldsymbol{x}$ is selected and the parameters are modified **greedily** so that $f(\boldsymbol{x}; \theta)$ is better on this very specific image. Following this *tug-of-war* dynamic, over time the parameters end up at some middle ground where f is good for all images.

However, notice what would happen if instead you split the dataset in 10 groups of 10 classes and sequentially train until convergence on each of those (like the in Split CIFAR100 dataset). The parameters would travel from one equilibrium to an other, every time decreasing performance for the previous classes because the associated images stop applying pressure to the optimization. We would say that the model has suffered from *catastrophic interference*, or negative backward transfer.

In more formal terms, in naïve continual learning we are sequentially solving

$$\theta_t = \arg\min_\theta l(f(\boldsymbol{X}_t; \theta), \boldsymbol{Y}_t)$$

and carrying the parameters every time while our goal is in fact is in fact the following

$$\theta_{ML} = \arg\min_\theta l(f(\boldsymbol{X}; \theta), \boldsymbol{Y})$$

This is not formally justified and clearly wishful thinking since previous tasks have to ways of informing future tasks. We have to find ways to restrict new tasks to solutions that are also adequate for previous tasks. We describe the three approaches usually considered.

3.1. Replay-based approaches

One popular class of approaches to solve this problem is to save some representative samples from every tasks and re-inject them into the training of future task (De Lange & Tuytelaars, 2021; Lopez-Paz & Ranzato, 2017; Rebuffi et al., 2016; Rolnick et al., 2018). Alternatively, a generative model can be trained to generate samples from past task (Shin et al., 2017), but the idea remains the same. This in some sort side steps the problem at hand by making old tasks samples apply pressure to the new task optimization (i.e. making the data more i.i.d.).

While these methods achieve very good performance in practice and are additionally justified by evidence of replay in the brain (van de Ven et al., 2020), they are probably not the end all solution. On one hand, the number of samples that can be replayed is limited for efficiency reasons but on the other hand using too few samples can lead the model to overfit to them, which is often observed. As the number and diversity of tasks increases, the number of samples required to represent all that should not be forgotten is going to be too large. Moreover, more practically, sometime storing past data is just not possible (due to privacy reason or other technical reasons).

3.2. Regularization-based approaches

Instead of trying to get back to the frequentist maximization of the likelihood with i.i.d. data, we can accept our fate and look for a principled way of optimizing (1) sequentially. The baysian perspective offers just that : maximize the posterior distribution which can be rearanged in the following way because the different datasets are assumed independent :

$$\begin{aligned} p(\theta \mid \mathcal{D}) &\propto \overbrace{p(\mathcal{D} \mid \theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}} \\ &\propto p(\mathcal{D}_t \mid \theta) p(\mathcal{D}_{1:t-1} \mid \theta) p(\theta) \\ &\propto \underbrace{p(\mathcal{D}_t \mid \theta)}_{\text{likelihood}} \underbrace{p(\theta \mid \mathcal{D}_{1:t-1})}_{\text{prior}} \end{aligned}$$

This gives a us a principle way to think about CL as optimizing each new task with the posterior of previous tasks as a prior. Notice that if we take the negative log we obtain the following optimization problem

$$\arg\min_\theta [-\log p(\mathcal{D}_t \mid \theta) - \log p(\theta \mid \mathcal{D}_{1:t-1})]$$

which can be interpreted as the classic loss minimization problem with a regularization term corresponding to prior :

$$R(\theta) = -\log p(\theta \mid \mathcal{D}_{1:t-1})$$

This is the intuition behind regularization-based methods. Note that sometimes it is taken loosely and the regularization term used is not formally related to the bayesian posterior, but still acts a proxy for previous tasks. The general formulation could be the following :

$$\underbrace{\nabla_\theta (l(y, \hat{y}) + R(\theta))}_{\text{New update}} = \underbrace{\nabla_\theta l(y, \hat{y})}_{\text{Go down gradient}} + \underbrace{\nabla_\theta R(\theta)}_{\text{But avoid interference}}$$

We quickly go over the most important regularization-based methods because they will be relevant later.

3.2.1. IMPORTANCE-BASED REGULARIZATION (LAPLACE PRIOR)

The most important (pun intended) line of work on regularization-based methods originates form **Elastic Weight Consolidation** (Kirkpatrick et al., 2016). It proposes to approximate the posterior $p(\theta \mid D_1, \dots, D_{t-1})$ with a diagonal covariance Gaussian bump around θ^*_{t-1}, the solution found at task t. Without going into the details, this gives the following approximation :

$$R(\theta) = -\log p(\theta \mid D_1, \dots, D_{t-1}) \approx \underbrace{\sum_{i=1}^{N}}_{\text{Diagonal}} \underbrace{\left(\sum_{\tau=1}^{t-1} \omega_i^\tau\right)}_{\text{Inverse of } \sigma_i^2} \underbrace{(\theta_i - \theta_i^{t-1})^2}_{\text{Centered around } \theta^{t-1}}$$

where

$$\omega_i^t = \mathbb{E}_{\boldsymbol{x}\sim\mathcal{X}_\tau, y\sim f_\tau(\boldsymbol{x};\theta)} \left[\left(\frac{\partial l(f_\tau, \mathcal{D}_\tau; \theta)}{\partial \theta} \right)^2 \Big|_{\theta^\tau} \right] \tag{2}$$

$$= \underbrace{\mathbb{E}_{\boldsymbol{x}\sim\mathcal{X}_\tau, y\sim f_\tau(\boldsymbol{x};\theta)} \left[\left(\frac{\partial \log f_\tau(\boldsymbol{x};\theta)_{\hat{y}}}{\partial \theta} \right)^2 \Big|_{\theta^\tau} \right]}_{\text{How sensible is the prediction to } \theta \text{ on task } \tau} \tag{3}$$

Pragmatically, adds a quadratic penalty to the parameters that move away from the mean of Gaussian bump (i.e. θ^{t-1}) with a weight that is based on the inverse of the variance of the approximation in that dimension : the more the approximation is confident for θ_i, the more it is important. For this reason, they call the term $\omega_i^\tau > 0$ *importance* of parameter θ_i in task τ.

$$R(\theta) = \sum_{i=1}^{N} \left(\sum_{\tau=1}^{t-1} \underbrace{\omega_i^\tau}_{\text{Importance}} \right) \underbrace{(\theta_i - \theta_i^{t-1})^2}_{\text{Quadratic penalty}}$$

Note that the importance is intuitively satisfying on its own : the more past prediction are sensitive to the value of θ_i (i.e the more $\frac{\partial l(y,\hat{y})}{\partial \theta}$ is big), the less it should change in the future. In fact, based on that, two other works have proposed different importance measures ω_i^τ that have different properties. Interestingly, (Benzing, 2020) has shown that they can both be seen as approximating the variance (or something close to it) of the Laplace approximation, like in EWC.

Synaptic Intelligence (Zenke et al., 2017) proposes an importance measure that can be computed online whereas the EWC importance is computed offline after each task. Specifically, at each update and for every parameter, they track the contribution of the parameter to the change in loss and accumulate this quantity over all the updates :

$$\tilde{\omega_i}^t = \sum_{j=0}^{M-1} \frac{\partial \mathcal{L}(j)}{\partial \theta_i} \cdot (\theta_i(j+1) - \theta_i(j))$$

Note that this is approximately computing the path integral of the change in loss :

$$\begin{aligned} \mathcal{L}(0) - \mathcal{L}(M) &= -\int_{\theta(0)}^{\theta(M)} \frac{\partial \mathcal{L}(j)}{\partial \theta} \cdot \theta'(j) dj \\ &= -\sum_{i=1}^{N} \int_{\theta_i(0)}^{\theta_i(M)} \frac{\partial \mathcal{L}(j)}{\partial \theta_i} \cdot \theta_i'(j) dj \\ &\approx -\sum_{i=1}^{N} \tilde{\omega}_i^t \end{aligned}$$

Finally, the importance is rescaled since the change in loss might be quite different from task to task and clipped because be only care about the decrease of the loss :

$$\omega_i^\tau = \frac{\max\{0, \tilde{\omega}_i^\tau\}}{(\theta_i(M) - \theta_i(0))^2 + \xi}$$

Notice the constant ξ, a damping hyper-parameter introduced for stability. A potential drawback of this method is that you cannot compute the importance of parameters in a pre-trained network (it has to be computed online with the gradient updates).

Memory Aware Synapses (Aljundi et al., 2017), proposes a heuristically derived importance measure that is also online and interestingly doesn't require ground truth labels to be computed. Instead of the loss' sensitivity to parameters like in EWC, MAS simply considers the sensitivity of the output of the network:

$$\omega_i^\tau = \mathbb{E}_{\boldsymbol{x}\sim\mathcal{X}_\tau} \left[\left| \frac{\partial \| f_\tau(\boldsymbol{x};\theta) \|^2}{\partial \theta_i} \right| \right]$$

Note that this importance measure is tightly related to biologically plausible mechanisms of synaptic plasticity regulation in the brain.

All these methods can be understood as approximating the posterior of the parameters for previous tasks with a diagonal Gaussian bump around θ_{t-1} and using this as a prior for optimization at task t. Notice that this approximation is only good in the neighborhood of θ^{t-1}, which can be a big problem. Indeed, this restricts new solution to stay close to old one **in the space of parameters**, which is not necessarily a good idea : the parameters that are optimal for task 1 and 2 can be very far from the parameters optimal for task 1 alone. This is especially true for deep neural networks which tend to have very task-specific features that use up a lot of parameters.

Therefore, each subsequent models is forced to build on the entangled features found in previous tasks and this can lead to the network quickly loosing capacity and becoming figuratively "stuck" in parameter space. The representations become clunky because the model must carry the dead weight of all the parameters that have been stabilized.

Overall, in an attempt to stabilize the past knowledge, regularization methods often greatly restrict plasticity.

3.2.2. Other approaches

A growing body of work in Bayesian Deep Learning uses a variation approximations for the posterior and obtains similar results as the Laplace approximation described in the last section (Farquhar & Gal, 2019; Lee et al., 2017; Nguyen et al., 2017). We don't go into this since this is not the focus of our work.

Another class of work, often called distillation-based methods, regularizes the new model to stay close to the old model by enforcing the activation of the two to be close on some unlabeled data (Jung et al., 2017; Li & Hoiem, 2016). For example, the activations can be compared on random noise, unlabeled data or the data of the new task. While the regularization term here is not formally related to the bayesian posterior, it still enforces the solution for the new task to be similar to previous ones.

3.3. Architectural approaches

Similar to regularization methods, architectural solutions apply a prior on the solution of new task by stabilizing parameters from previous tasks. However, they do so in much more controlled manners that aim at balancing the stability and plasticity of the models. In particular, they force each task to only use a small number of parameters (i.e. have sparse, compact representations) so that stabilizing them won't hamstring the capacity of subsequent models.

The main questions they have to answer are

- How to find compact representations for the tasks?
- How to stabilize the parameters? (either freezing or regularization)
- How to reuse the representations of previous tasks?

We go in detail into the different ways of doing that in the next section.

4. Architectural approaches

We classify the existing along two main axis :

- **Fixed vs Expanding** : one can either start with a large model and progressively assign part of it to different tasks or start with a small model and progressively add parameters when needed.
- **Monolithic vs Modular** : models can either be normal monolithic deep learning models or modular networks made of several modules that can be composed. In the first case each task is assigned a subnetwork and in the second they are assigned certain modules.

Our main sections follow the Monolithic vs Modular axis.

4.1. Monolithic architecture

Here, every tasks are assigned a subnetwork, which is then frozen or regularized for subsequent tasks. Subsequent tasks can operate over the representations of previous tasks like in regularization-based methods, however, mechanisms are in place to make sure capacity doesn't saturate quickly.

4.1.1. Fixed monolithic architecture

In fixed monolithic architecture, the model start with a large capacity and must find/train task specific subnetworks within it. These approaches can be seen as progressively "filling" the model, but in a controlled way that leaves capacity to all tasks.

First, to find a small subnetwork for a task **PackNet** (Mallya & Lazebnik, 2017), proposes to use a simple pruning algorithm which trains the full network, removes the smallest $50\% - 75\%$ weights and then retrains the resulting subnetwork. For the first task, this procedure is ran on the full network and the subnetwork (identified by a mask M_1) is frozen. Then, the next task repeats the same procedure but does the pruning only on the remaining weights (so subsequent tasks use less and less new parameters). Importantly, it reuses the whole subnetwork previously frozen so that $M_t \subseteq M_{t+1}$. Also to solve task t at inference, the mask M_t is applied to the network so that later tasks don't interfere with the representation learned during training.

Similarly, **Continual Learning via Neural Pruning (CLNP)** (Golkar et al., 2019) assigns iteratively larger subnetwork to every task but uses a different pruning procedure. In particular, instead of pruning based on the magnitude of the weights, they identify *neurons* whose average activation on a task exceeds some threshold and freezes/masks the subnetwork induced by these neurons. To illustrate the difference, in a CNN, PackNet would freeze some weights throughout the kernels while CLNP would freeze some kernels in their entirety. Pruning neurons instead of weights has many advantages : stored masks are more light-weight, matrix multiplications can be sped by removing zeroed-out dimension (neurons) and new features are not constrained by some of the weight being fixed due to previous tasks.

Instead of training the full network, pruning and then re-

training (which can be quite expensive), **Hard Attention to the Task (HAT)** (Serrà et al., 2018) develops a way to directly learn the subnetwork for each task. To do that, at task t, the units are multiplied element-wise by a soft mask $\boldsymbol{M}^t = \sigma(s\boldsymbol{E}^t)$ where s is a constant controlling the steepness of the sigmoid and $\boldsymbol{E}^t$ is a learnable task embedding. Moreover, during training, they add a regularization term encouraging $\boldsymbol{M}^t$ to be sparse. This way, the network learns a (soft) sparse subnetwork for each task end-to-end with gradient descent. Importantly, when learning task t, the gradient of the parameters are multiplied by $1 - \max\{\boldsymbol{M}^1, \dots, \boldsymbol{M}^{t-1}\}$ where the $\max$ is element-wise to protect previous subnetworks from interference. It outperforms previous both in performance and in efficiency. **Relevance Mapping Networks** operate in an extremely similar way.

Context-Dependent Gating (XdG) (Masse et al., 2018) avoid completely the problem of learning the structure of the subnetworks by fixing each task's subnetwork at initialization. Specifically, the mask for every task is induced by fixing the activation of 80% randomly chosen neurons to 0 (with potential overlap between tasks). Also, instead of freezing after the subnetwork after the task training is finished, XdG uses importance-based regularization like EWC and SI. The regularization protects the overlapping neurons from interference while restricting the different task to different subnetworks prevents regularization from stabilizing too many neurons. Indeed, they show that even after a lot of tasks, much less neurons have high importance weights than if regularization was used alone. Because of this, XdG yields much better performance than regularization alone (and similar performance as RMN). This simple technique illustrates very well the weakness of regularization and the usefulness of architectural methods. The main drawback of XdG is that is makes positive forward transfer very unlikely because weights are shared in a random way.

Like in XdG, the **Active Dendrites Network** (Iyer et al., 2022) aims at activating different sparse and weakly overlapping subnetworks for every task in an attempt to avoid catastrophic forgetting. However, instead of assigning random and fixed subnetworks to every task, the subnetworks are learned through a brain-inspired mechanism. Similar to HAT, the units are multiplied element-wise by a soft-mask $\boldsymbol{M}^t = \sigma(s\boldsymbol{D}^t)$. However instead of being a simple learnable task embedding, $D_i^t = \max_j \langle \boldsymbol{u}_j^{(i)}, \boldsymbol{c}_t \rangle$ where $\{\boldsymbol{u}_j^{(i)}\}_i$ are the *dendritic segments* of neuron i and $\boldsymbol{c}_t$ is a *context* associated with task t. Without loss of generality, the segments and context are random small vectors. Intuitively, when at least one of the dendritic segments has a strong response to the context (i.e. large dot product), then the neuron will be active. Moreover, after each layer of the modified MLP, only a very small number k of neurons with the highest activation are sent to the next layer, the rest are multiplied by 0 (a dynamic supposedly mimicking biological inhibitory networks). These two architectural components have the following effect : only the neurons that have high dendrite activation will be able to ever win the competition and therefore be part of the subnetwork corresponding to the context (i.e. the task). Like in XdG, regularization is used. It obtains similar performance as XdG but because the subnetworks are learned, it supports positive forward transfer and uses the capacity dynamically and therefore more economically. The idea of gating neurons activity with neuron activity has recently gotten more experimental support with the work of (Russin et al., 2022). The potential of this approach is still largely unexplored as it was tested on very limited benchmarks.

Finally, in a somewhat unconventional way **Supermasks in Superposition (SupSup)** (Wortsman et al., 2020) proposes to do the opposite of XdG : freeze all the weights and learn only the structure of the subnetworks. This is inspired by the work on the Lottery Ticket Hypothesis which argues that a randomly initialized possesses special subnetworks–called supermasks–which acheive good performance on random tasks. Therefore, for every task, a subnetwork with good performance is found with the weight-picking algorithm of (ramanujan) and the corresponding mask is associated with the task. Since the weights cannot change, catastrophic interference is completely avoided. However, this also means that there can be no transfer of knowledge : as indicated by the title this only aims at superposing models. Note that **Piggyback** does essentially the same thing as SupSup but starting with a pre-trained neural network (e.g. ResNet-18 on ImageNet).

4.1.2. Expanding monolithic architecture

In the expanding monolithic architecture the model starts out small and for every new task some amount of new neurons are added and assigned to the task. This has the advantage that it can in theory support an unlimited amount of tasks. However, the task of figuring out how many neurons to add is often complicated and computationally expensive.

Progressive Neural Networks (PNN) (Rusu et al., 2016) is the first model to consider model expansion in the context of continual learning. In a PNN, every time a new task is encountered a new identical MLP is trained (called a *column*). Importantly, previous columns are frozen at the beginning of a new task and the new network has lateral connection to all previous columns at every layer. More precisely, the activations at layer l of all previous columns go through a one layer MLP and is fed to layer $l+1$ of the new network. The main drawback of PNNs is that a lot of parameters are added for every task, even it might not be needed depending on the similarity of the tasks and on how many tasks have already been seen.

To solve this problem, **Reinforced Continual Learning (RCL)** (Xu & Zhu, 2018) proposes a mechanism to add an adaptive number of neuron at every layer of the new column. At every task, instead of adding the same fixed number of neurons to every layer, they train a *controller* to find the best amount. Specifically, the controller is an RL agent producing a series of actions $\boldsymbol{a}_{1:L}$ which are interpreted as the number of neurons to add at every layer and whose reward is

$$R = A(\boldsymbol{a}_{1:L}) + \alpha \sum_{i=1}^{L} a_i$$

where A is the accuracy obtained by network augmented $\boldsymbol{a}_{1:L}$ neurons and the regularization term encourages only a few neurons to be added. The agent is implemented in a unconventional way with an autoregressive LSTM always starting from the same input. In order to train the controller on a given task, the network has to be retrained a large number of time with varying number of added neurons which is extremely time consuming. Although it leads to less memory consumption, RCL needs up to 80x compute in order to get the same performance as PNN.

Similarly, **Dynamically Expendable Networks (DEN)** (Yoon et al., 2017) propose an elaborate network expansion scheme that is more efficient than the greedy one in PNNs. In particular, for every new task, the following steps are taken :

1. Train the old weights θ^{t-1} on new task to obtain θ^t
2. If the loss is above some threshold
   (a) Add a column like in PNNs to get $[\theta^t, C]$
   (b) Train C with θ^t frozen and with a group regularization term that encourages only a few neurons to have non-zero weights.
   (c) Remove neurons that got killed by the regularization to get $[\theta^t, \bar{C}]$
3. If the incoming weights of a neuron have changed beyond a certain threshold from θ^{t-1} to θ^t, reintroduce a copy of the old neuron to get $[\theta^t, \bar{C}, \bar{\theta}^{t-1}]$. This means that the important weights from last task are either hardly modified or preserved, a slightly more parameter efficient than just freezing.
4. Retrain $[\theta^t, \bar{C}]$ and set $\theta^t = [\theta^t, \bar{C}, \bar{\theta}^{t-1}]$

Additionally, for technical reasons, they use sparse weights throughout. Again, the important drawback of this technique is its complexity and high number of hyperparameters : it need up to about 50x more compute than PNN to achieve the same performance.

Finally, **Compact, Pick and Grow (CPG)** (Hung et al., 2019) devises a multi-step adaptive network expansion based on pruning. On the first task, like in PackNet, the network is trained, pruned of its unnecessary weights and then frozen. Then, the free weights are trained for the next task but unlike in PackNet not all the frozen weights are used by the new task. Instead, while learning the new task on the free weights, a weight-picking mask like the one in SupSup is trained on the frozen weights so that only a few (the most important) are used. The motivation behind that is that after a lot of tasks there is a lot of frozen weights and it can make learning how to use them difficult. Then also unlike PackNet, if the resulting network for the new task doesn't attain a certain level of performance, new weights are added (e.g. new filters, wider MLP) and training continues. Finally, the new model is pruned and frozen and the process starts over again. While their algorithm is overall simpler than DEN and RCL, it still requires much more compute than PNN to obtain the same performance. However, much less so than DEN and RCL and it actually outperforms the three previous on complex tasks.

4.2. Modular architecture

Here, the neural network is made of modules through through which the input is routed by a *router*. The input goes through a certain number of modules at every layer and then the different outputs are summed together to form the output for the next layer. This is for architectural methods because it gives a simple way of assigning different part of the model to different tasks. Also modular neural networks are know to form more factorized representations that are useful in continual learning (have higher chances of being relevant for the next tasks) (Rosenbaum et al., 2019).

There are two types of modular architectures depending on the nature of the routing : hard routing or soft routing. In hard routing, the router makes a discrete decision of sending the input to a certain number of module per layer. In soft routing, the input is sent to every module but the output of every module is weighted by the router.

4.2.1. Fixed modular architecture

First, **PathNet** (Fernando et al., 2017) considers a hard routing network made of L layers of M modules. For every task, PathNet will learn which modules (a maximum of $N < M$ at every layer) to activate and train their weights at the same time. To solve this complex problem, authors use an evolutionary algorithm : a pool of initially random pathways (defining which modules to use) are all trained with gradient descent for a few epochs and the winning pathway P^* gets to repopulate the pool with randomly altered copy of itself (i.e. with some modules changed). After this process goes on for some time, the winning pathways is assigned to the task and is frozen. The obvious drawback of this approach is the cost of the evolutionary procedure.

On the other hand, **Dynamic Information Balancing (DIB) Routing** (Amer & Maul, 2019) uses a normal dynamical routing network where the input is only routed through one module per layer, decided by a reinforcement learning agent at every layer whose actions are the different modules (rosenbaum). The only addition that this paper makes in regards to continual learning is the addition of regularization (EWC) and they get much better performance than EWC or routing alone. Indeed, routing network by themselves tend to make representations more disentangled, thus it positively interacts with regularization.

4.2.2. Expanding modular architecture

Here, to make the combination search space for the task-specific path smaller, models start with a small number of modules and slowly add more.

Similar to PathNet, **Learn-to-grow (LTG)** (Li et al., 2019) performs a hard selection of modules based on the task. However, unlike PathNet 1) each task uses only one module per layer and 2) the pool of modules is gradually expanded, starting with one per layer. At layer l the path of a new task will either : 1) *reuse* an already existing module, 2) go through *adapted* version of an old module 3) go through an entirely new module. Here *adapted* means that the old module is copied, frozen and modified by few parameters (e.g. for a convolution layer the adaptation is adding a 1x1 convolution alongside it). This creates a space of possible new path, which is searched through by a first-order DARTS : a model where each layer is a weighted sum of all the possible choices $x^l = \sum_i \alpha_i^l m_i^l(x^{l-1})$ is instantiated and trained by alternatively updating $\alpha = \{\alpha^l\}$ and the modules weights. After some time, the path obtained by taking the index of the largest weight α_c^l is chosen and further optimized. The added modules, if any, are added to the pool and frozen for the rest of tasks.

Overall, **Modular Networks with Task-Driven Priors (MNTDP)** (Veniat et al., 2020) is overall extremely similar to LTG as it learns a one-module-per-layer path for every task building on the previous ones. In particular, after the first task, the following steps are taken at every task :

1. One new module per layer is initialized
2. Out of the paths taken by all previous tasks, the one that performs best on the task is selected.
3. The best combination of the selected path and new modules (selecting either old path or new module at every layer) is found and associated with the new task. The new modules that were chosen are added the the pool and are frozen.

For the step 3, the authors propose two ways of finding the best combination. The first (deterministic) is to train in parallel all the possible combination and choose the best. The second (stochastic) is to alternate gradient updates between learning a distribution over combinations and learning a path sampled form that distribution. Because it only searches how to alter to previous best path, this technique is more efficient than LTG, but it still achieves similar performance.

Another approach to learning the module selection, proposed by **Lifelong Learning of Compositional Structures (LLCS)** (Mendez & Eaton, 2020), is to use soft and learnable module selection at every layer, i.e. the output of layer l is:

$$\boldsymbol{x}^{(l)} = \sum_{m_i \in \mathcal{M}_l} [\boldsymbol{s}_\psi(\boldsymbol{x}^{(0)}, t, l)]_i \cdot m_i(\boldsymbol{x}^{(l)})$$

where $\mathcal{M}_l$ are the modules at layer l, $\boldsymbol{x}$, t is the task and $\boldsymbol{s}_\psi(\boldsymbol{x}, t, l)$ is a *routing function* such that sums to 1 for every layer. This way, we can learn which modules to use for a given task/input end-to-end with gradient descent. give a general procedure to use such architecture in the context of CL. Their procedure is split into three steps : $\boldsymbol{s}_\psi$

1. Initialization : $\boldsymbol{s}_\psi$ and $\mathcal{M}$ is randomly initialized. Then $\boldsymbol{s}_\psi$ is frozen and the first T_{init} tasks are learn jointly only updating the modules.
2. Assimilation : As the first phase of training for every task $t > T_{init}$, modules are frozen and only $\boldsymbol{s}_\psi(t)$ is updated.
3. Accommodation : As the second phase, $\boldsymbol{s}_\psi(t)$ is frozen and only modules are learned. Importantly, regularization is used to prevent catastrophic forgetting.

To augment their fixed-architecture version, the paper of the last section also proposed an expansion strategy very similar to the one in MNTDP (deterministic). Specifically, they perform the Assimilation phase many times : in each, one new randomly initialized module is added to one of the layers and trained along with the routing function (unlike other modules). Then, the best performing layout is kept if it improves performance by more than a threshold τ. Note that during the new assimilation phases the weight of the new module is fixed to 1 to ensure that it is used. Similar to DIB routing, the only thing that stop catastrophic interference is using a regularization technique and it strongly outperforms regularization alone. Note that the expansion strategy is very inefficient, however it was not their main focus and could easily be improved.

Finally, **Continual Learning via Local Module Composition (LMC)** (Ostapenko et al., 2021) uses soft module selection and progressively adds more modules but in a different way that makes the task ID not needed during inference. Specifically, to compute the weighting of the different modules for input $\boldsymbol{x}$ and layer l, each module locally

computes a relevancy score $\gamma_m^{(l)}(m_i^{(l)}(\boldsymbol{x}^{(l-1)})$ and all the scores compete together from $\boldsymbol{w}^{(l)} = \text{softmax}(\boldsymbol{\gamma}^{(l)})$:

$$\boldsymbol{x}^{(l)} = \sum_{m_i \in \mathcal{M}^{(l)}} w_m^{(l)} \cdot m_i(\boldsymbol{x}^{(l-1)})$$

In lower layers, the relevancy score is the negative loss of a decoder trained to reconstruct $\boldsymbol{x}^{(l-1)}$ and in higher layers it the negative norm of a normalizing flow taking $m_i^{(l)}(\boldsymbol{x}^{(l-1)})$ as input. At first, the network starts with one module by layer and the addition of a module is triggered at a certain layers l when the input $\boldsymbol{x}^{(l-1)}$ is not relevant to any modules (i.e. the modules were not trained to deal with it). When a new module is added, for a fixed number of epochs, the new module is trained to maximise the relevance score of higher modules (along with the normal loss) and no other module can be added. Similar to PathNet, catastrophic interference is avoided by freezing all modules at the end of every task. Note that module of the last layer the output head and a new one is used added at every task during training. However, at inference, they are all used and the one with the highest relevance score is selected (therefore not requiring a task ID).

5. Discussion

We have seen how naïvely using classic deep learning methods in the continual learning framework leads to some major problem, notably catastrophic interference. Then, we have introduced in a principled way the different approaches commonly used to resolve this problem and how they relate. In particular,

- the replay methods sticks to the frequentist viewpoint and tries to get back to the i.i.d. scenario that works
- the regularization-based methods instead considers the bayesian viewpoint and formulate a prior based on previous tasks solutions to guide the optimization of new task
- the architectural methods also constraint the optimization of new task, but in a much more controlled way to avoid the clunkyness and quick loss of capacity that regularization-based methods exhibit

We focused on the latter class of methods because there is currently no survey dedicated to them, they tend to be confusing and we believe they are importance. Indeed, we argued that they are necessary in order to achieve the plasticity-stability tradeoff : synaptic stabilization alone on current deep neural networks inevitably leads to clunky representations and in turn to quick capacity saturation. By making the representations more sparse and limiting the localizing the effect of stabilization, architectural methods lead to way to a scalable continual learning. Our goal was to describe in the most simple way possible the space of possible tricks that currently exist to understand current the current state of the field and guide future works. We quickly summarize the different groups of methods we consider :

- **Fixed monolithic architectures** try to learn every task into a sparse subnetwork, which is then stabilized and potentially reused by other tasks. The subnetwork can be found by pruning, learned end-to-end or randomly selected.
- **Expanding monolithic architectures** progressively add new parameters, learn a task into those parameters and then freeze every. This avoid the having to find a subnetwork since all the new parameters can be used. However, this creates a problematic tradeoff regarding scaling: deciding how many new parameters to add of a new task can be very computational expensive while adding too much can be learn to memory explosion.
- **Modular architectures** have a compositional structure that makes them especially good at learning disentangled representations hence they are generally good for continual learning and they positively interact with regularization techniques. However, some of them tend to be very computationally expensive, but it's exactly say how much because they do not discuss it.

We notice that a common pitfall of current techniques is that they tend to be highly engineered and thereby incur a lot of computational overhead. We would prioritize over such methods the study of simple ideas that rest on simple principles. Moreover, we think that future works clearly measure the computational footprint of their architecture and compare it with others, something that is rarely done and would encourage further clever and simple solutions. Even better, a clear experimental comparison of the different approaches discussed here would be very useful, and a potential future direction for this work.

References

Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., and Tuytelaars, T. Memory aware synapses: Learning what (not) to forget, 2017. URL https://arxiv.org/abs/1711.09601.

Amer, M. and Maul, T. Reducing catastrophic forgetting in modular neural networks by dynamic information balancing, 2019. URL https://arxiv.org/abs/1912.04508.

Benzing, F. Unifying regularisation methods for continual learning, 2020. URL https://arxiv.org/abs/2006.06357.

De Lange, M. and Tuytelaars, T. Continual prototype evolution: Learning online from non-stationary data streams. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 8250–8259, October 2021.

Delange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021. doi: 10.1109/tpami.2021.3057446. URL https://doi.org/10.1109%2Ftpami.2021.3057446.

Farquhar, S. and Gal, Y. A unifying bayesian view of continual learning, 2019. URL https://arxiv.org/abs/1902.06494.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. Pathnet: Evolution channels gradient descent in super neural networks, 2017. URL https://arxiv.org/abs/1701.08734.

Golkar, S., Kagan, M., and Cho, K. Continual learning via neural pruning, 2019. URL https://arxiv.org/abs/1903.04476.

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. An empirical investigation of catastrophic forgetting in gradient-based neural networks, 2013. URL https://arxiv.org/abs/1312.6211.

Hung, S. C. Y., Tu, C.-H., Wu, C.-E., Chen, C.-H., Chan, Y.-M., and Chen, C.-S. Compacting, picking and growing for unforgetting continual learning, 2019. URL https://arxiv.org/abs/1910.06562.

Iyer, A., Grewal, K., Velu, A., Souza, L. O., Forest, J., and Ahmad, S. Avoiding catastrophe: Active dendrites enable multi-task learning in dynamic environments. *Frontiers in Neurorobotics*, 16, apr 2022. doi: 10.3389/fnbot.2022.846219. URL https://doi.org/10.3389%2Ffnbot.2022.846219.

Jung, H., Ju, J., Jung, M., and Kim, J. Less-forgetful learning for domain expansion in deep neural networks, 2017. URL https://arxiv.org/abs/1711.05959.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., and Hadsell, R. Overcoming catastrophic forgetting in neural networks, 2016. URL https://arxiv.org/abs/1612.00796.

Larochelle, H., Erhan, D., Courville, A., Bergstra, J., and Bengio, Y. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pp. 473–480, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937933. doi: 10.1145/1273496.1273556. URL https://doi.org/10.1145/1273496.1273556.

Lee, S.-W., Kim, J.-H., Jun, J., Ha, J.-W., and Zhang, B.-T. Overcoming catastrophic forgetting by incremental moment matching, 2017. URL https://arxiv.org/abs/1703.08475.

Li, X., Zhou, Y., Wu, T., Socher, R., and Xiong, C. Learn to grow: A continual structure learning framework for overcoming catastrophic forgetting. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3925–3934. PMLR, 2019. URL http://proceedings.mlr.press/v97/li19m.html.

Li, Z. and Hoiem, D. Learning without forgetting, 2016. URL https://arxiv.org/abs/1606.09282.

Lopez-Paz, D. and Ranzato, M. Gradient episodic memory for continual learning, 2017. URL https://arxiv.org/abs/1706.08840.

Mallya, A. and Lazebnik, S. Packnet: Adding multiple tasks to a single network by iterative pruning, 2017. URL https://arxiv.org/abs/1711.05769.

Masse, N. Y., Grant, G. D., and Freedman, D. J. Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization. *Proceedings of the National Academy of Sciences*, 115(44):E10467–E10475, 2018. doi: 10.1073/pnas.1803839115. URL https://www.pnas.org/doi/abs/10.1073/pnas.1803839115.

Mendez, J. A. and Eaton, E. Lifelong learning of compositional structures, 2020. URL https://arxiv.org/abs/2007.07732.

Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. Variational continual learning, 2017. URL https://arxiv.org/abs/1710.10628.

Ostapenko, O., Rodriguez, P., Caccia, M., and Charlin, L. Continual learning via local module composition. 2021. doi: 10.48550/ARXIV.2111.07736. URL https://arxiv.org/abs/2111.07736.

Rebuffi, S.-A., Kolesnikov, A., Sperl, G., and Lampert, C. H. icarl: Incremental classifier and representation learning, 2016. URL https://arxiv.org/abs/1611.07725.

Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T. P., and Wayne, G. Experience replay for continual learning, 2018. URL https://arxiv.org/abs/1811.11682.

Rosenbaum, C., Cases, I., Riemer, M., and Klinger, T. Routing networks and the challenges of modular and compositional computation, 2019. URL https://arxiv.org/abs/1904.12774.

Russin, J., Zolfaghar, M., Park, S. A., Boorman, E., and O'Reilly, R. C. A neural network model of continual learning with cognitive control, 2022. URL https://arxiv.org/abs/2202.04773.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. Progressive neural networks, 2016. URL https://arxiv.org/abs/1606.04671.

Serrà, J., Surís, D., Miron, M., and Karatzoglou, A. Overcoming catastrophic forgetting with hard attention to the task, 2018. URL https://arxiv.org/abs/1801.01423.

Shin, H., Lee, J. K., Kim, J., and Kim, J. Continual learning with deep generative replay, 2017. URL https://arxiv.org/abs/1705.08690.

van de Ven, G. M., T Siegelmann, H., and S Tolias, A. Brain-inspired replay for continual learning with artificial neural networks, 2020. URL https://pubmed.ncbi.nlm.nih.gov/32792531/.

Veniat, T., Denoyer, L., and Ranzato, M. Efficient continual learning with modular networks and task-driven priors, 2020. URL https://arxiv.org/abs/2012.12631.

Wortsman, M., Ramanujan, V., Liu, R., Kembhavi, A., Rastegari, M., Yosinski, J., and Farhadi, A. Supermasks in superposition, 2020. URL https://arxiv.org/abs/2006.14769.

Xu, J. and Zhu, Z. Reinforced continual learning, 2018. URL https://arxiv.org/abs/1805.12369.

Yoon, J., Yang, E., Lee, J., and Hwang, S. J. Lifelong learning with dynamically expandable networks, 2017. URL https://arxiv.org/abs/1708.01547.

Zenke, F., Poole, B., and Ganguli, S. Continual learning through synaptic intelligence, 2017. URL https://arxiv.org/abs/1703.04200.