University of Padova

Department of Information Engineering

Master Thesis in Control Systems Engineering

Analysis of the impact of class ordering in class incremental image classification

Supervisor
Pietro Zanuttigh
University of Padova

Co-supervisor
Francesco Barbato
Marco Toldo

Master Candidate
Marwa Abdelkarim

Academic Year
2022–2023

Padova, 20/04/2023

Abstract

The benefits of incremental learning make it desirable for many real-world applications. It enables efficient utilization of resources by eliminating the need to start training from scratch when the considered set of tasks is updated. Additionally, it reduces memory usage, which is particularly important in situations where privacy limitations exist, such as in the healthcare sector where storing patient data for a long time is prohibited. However, the main challenge of incremental learning is catastrophic forgetting, which causes a decline in the performance of previously learned tasks after learning a new one. To overcome this challenge, various incremental learning methods have been proposed.
In this work, we explore the influence of class ordering on class incremental learning and the resilience of the method to different class orderings. Additionally, we examine how the complexity of incremental learning scenarios or task split strategies affects the model's performance. We start with a pre-existing approach and then introduce extensions to improve its performance.
Experimental results show that the model's performance is not too significantly impacted by the sequence in which classes are presented, but the complexity of the incremental tasks plays a crucial role in determining the model's performance. Additionally, starting with a higher number of classes typically results in better performance.

Acknowledgments

I express my deep gratitude to my supervisor, Professor Pietro Zanuttigh, for his guidance and support throughout my thesis journey. I also extend my thanks to my advisors, PhD students Francesco Barbato and Marco Toldo, for their help during the development and writing of my thesis.
Additionally, I appreciate the friendship and memories shared with my friends. Lastly, I owe this achievement to my family for their unwavering support and encouragement throughout my academic years.

Contents

Listing of figures

Listing of tables

1 Introduction

Deep neural networks (DNNs) have attained remarkable results in a variety of computer vision tasks such as image classification, object detection, and segmentation, and have become state-of-the-art in the field. Despite their impressive performance in standard offline learning, where all the training data is available in a single session, it remains challenging for them to learn continuously and avoid starting from scratch when new data is introduced. Continual Learning (CL) [1] or Lifelong Learning (LFL) refers to the scenario in which the model needs to learn from a continuous stream of data.
In most incremental learning (IL) scenarios, the model is presented with tasks in a sequential manner, where only the data from one task is available for learning in each session. After each training session, the learner (DNN) should be capable of performing all previously seen tasks. The biological inspiration for this learning model is clear, as it reflects how humans acquire and integrate new knowledge: when presented with new tasks to learn, we leverage knowledge from previous ones and integrate newly learned knowledge into previous tasks [2].

Incremental learning has gained attention in the last few years due to its application in solving several problems, such as memory restrictions for systems that cannot store all data and need to learn incrementally, data security/privacy restrictions for systems that learn from data that cannot be permanently stored, and sustainable ICT where the cost of retraining deep learning algorithms for every task update is high. Incremental learning offers a more computationally efficient solution that only requires the processing of new data when updating the system.

The main challenge in continual learning is that DNNs tend to forget previous knowledge when learning new tasks, this phenomenon is called catastrophic forgetting [3, 4, 5]. This phenomenon occurs when a neural network model is trained for a new task, causing it to forget its previous training for a different task. This results in a good performance for the new task, but poor performance for the older one. When a network is trained for a new task, its weights are adjusted to optimize performance for the new task. This can cause the weights to change significantly from their initial values to the point where they no longer capture the previous knowledge.
Another challenge is the stability-plasticity dilemma [6], where a learner needs to balance its ability to learn new information (plasticity) without forgetting the previously learned knowledge (stability).

In this work, we consider class incremental learning, which is one of the three scenarios of continual learning as defined in [1]. In class incremental learning, each task consists of a set of classes that is distinct from the classes in previous tasks. The main goal is to develop a unified classifier that can accurately classify all classes encountered at different stages, without relying on task identification during inference.
There are multiple solutions proposed to solve class incremental learning, including storing a fraction of old data for joint training and using deep generative models to generate pseudo-samples of previous classes. However, training big generative models is inefficient and they suffer from catastrophic forgetting. Another approach is to penalize future changes to important parameters, but this is only effective with multi-head classifiers and task identifiers available at inference and has shown poor performance in CIL scenarios.

The objective of this thesis is to investigate the impact of class ordering on class incremental learning and the robustness of the method to class ordering. We also analyze the effect of the complexity of incremental learning scenarios or split strategies on the performance of the model. We begin with the incremental learning method proposed in [7]. This method is a non-exemplar method that does not store examples from previous classes. Instead, it augments a prototype from each previously learned class in the feature space and utilizes self-supervised learning to develop comprehensive and transferable features. To evaluate the method's robustness to class ordering, we used two distinct class ordering strategies, random order, and superclass order (ordered by grouping together similar classes). Additionally, we trained and tested

the model on different sequences of tasks (1, 4, 5, 9, 01, 19, 20) and compared the results of each learning scenario using Cifar-100 dataset. The evaluation of the overall performance for each sequence of tasks involves two metrics. First, the average accuracy is calculated after reaching the last task. Second, the average forgetting on all previously learned classes is computed. Finally, we developed an additional loss to enhance the model's performance.

The thesis is structured in the following manner: Chapter 2 provides an introduction to the problem of image classification, and describes convolutional neural networks and existing CNN-based approaches to address this problem. Chapter 3 is dedicated to incremental learning and covers the fundamental challenges associated with it, along with the techniques used to overcome them. Chapter 4 presents the details of the proposed model, including its architecture and optimization strategies that are adopted for training. Chapters 5 and 6 are focused on presenting the experimental results obtained through the proposed model and drawing conclusions based on the findings.

2

Image Classification

This chapter introduces some background knowledge and theory about image classification tasks. The related or essential concepts are explained in more detail to help readers better understand the thesis, while the others are mentioned to complete the review.

2.1 Image classification

Image classification is one of the most essential tasks in computer vision. It is the process of assigning a label or class to an image using features or information extracted from them. Let C be a fixed set of classes, x an input image, image classification is the task that designs a model h that predicts the class $y \in Y$ to which the input image x belongs.

$$h(x) = y \tag{2.1}$$

Image classification is based on the assumption that the image depicts one or more features and that each of these features belongs to one of several distinct and exclusive classes [8]. The classes may be specified a prior by an analyst (as in supervised classification) or automatically clustered (i.e. as in unsupervised classification) into sets of prototype classes, where the analyst merely specifies the number of desired categories [8].

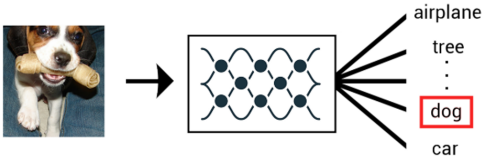


Figure 2.1: The classifier assign label to the input image from a fixed set of classes.

2.2 Supervised Learning

In supervised learning, the training dataset S is composed of pairs (input x, output y) and the goal is to learn a model that associate inputs with their outputs. In classification problems, the output is the set of classes or categories.
The learning process aims to determine the possible hypothesis (model $\hat{h}$) that best maps input to the corresponding output from the hypothesis space H. The hypothesis space is the set of all possible hypotheses that can map inputs to their outputs, it is defined prior based on the data and restrictions applied to the data.

$$H = \{h1, h2, ..h_{|H|}\} \tag{2.2}$$

The model $\hat{h}$ is the one with the minimum empirical loss. Empirical loss $L_s(h)$is the mismatch between model output (predicted) and true output ,defined on the training dataset S.

$$L_{s(h)} = \frac{1}{|S|} \sum_{(x,y)\in s} l(h(x), y). \tag{2.3}$$

$$\hat{h} = argmin_{h\in H} L_s(h) \tag{2.4}$$

The problem with hypothesis space is that, it may explode when the dataset has high number of features. To overcome this problem , artificial neural networks are introduced and the hypothesis space became fixed and represented by parameterized model.

$$H = \{M_\theta, \theta \in \Theta\} \tag{2.5}$$

Θ is the parameter space, and the best parameter is obtained by minimizing empirical loss.

$$\hat{\theta} = argmin_{\theta \in \Theta} L_s(M_\theta) \tag{2.6}$$

2.3 Artificial neural networks

In this subsection, we will introduce artificial neural networks (ANNs), which come in various forms and are used to solve a wide variety of tasks, including image classification. First, we will explain the ANN architecture, the algorithm used to optimize ANN models, and then we will describe in detail the convolutional neural networks (CNNs). CNN is more relevant to understand ResNet-18, the model we used to solve our task.

2.3.1 Artificial neural network architecture

Artificial neural network is inspired by biological neural networks and is presented as a combination of neurons that can calculate values based on the input. A neuron is defined as an information-processing unit that is fundamental to the operation of a neural network [9]. ANNs architecture compose of three main layers:

- Input layer which receives inputs to be processed.
- One or more hidden layers enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns [9]. For example, the first hidden layer can extract basic features as lines or edges from the input image, the next layer can learn more complex features as shapes.
- The output layer performs the required task such as classification or prediction.

ANN layer's are represented by several neurons (nodes) act in parallel. The neuron is the computing unit of the network, it receives input vector x and performs affine transformation (figure 2.3). The neuron output is computed by adding the sum of the inputs ($x1, ...xn$) multiplication by their corresponding weights ($w1, ...wn$) to the bias b, then the result fed to the activation function f. The weights w determines how the input of the previous layer will affect the output of the next layer, and the bias b helps to move the activation function to the positive and negative sides on the coordinate graph.

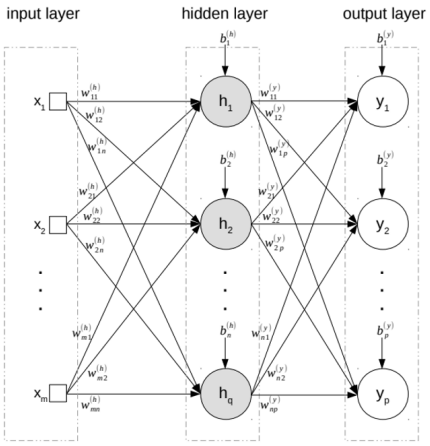


Figure 2.2: ANN architecture.

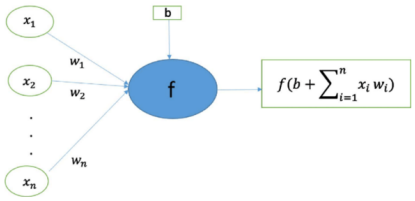


Figure 2.3: Neuron structure.

Activation function decides if the neuron will be activated or not. There is two types of activation function, linear and non-linear activation function. Non-linear activation functions help to increase the approximation quality of ANNs. Non-linear activation functions used in the thesis are:

- Leaky Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} x \text{ if } x > 0 \\ \text{Otherwise, } \ k.x \end{cases} \tag{2.7}$$

- SoftMax: Also known as the normalized exponential function It represents a probability distribution over a discrete variable with n possible values. It is used in the output layer nodes of a classifier as in our case.

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{2.8}$$

2.3.2 Convolutional neural networks

Convolutional neural network (CNN) is a class of ANN. CNN is more suitable for 2D structure data, such as images, because it has essential properties, Local connectivity, and spatial parameters. Local connection means each neuron is connected only to nearby neurons from the previous layers, and the latter property enables using the same filter by different neurons in the same layer, which reduces the required memory size and number of learnable parameters. CNN architecture composes of convolutional layers, pooling layers, and fully connected layers (figure 2.4). We will go into details of each layer below.

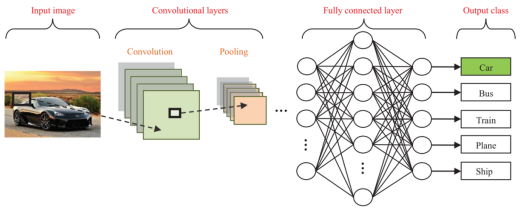


Figure 2.4: CNN image classification pipeline.

Convolutional layers act as feature extractors, learning the feature representations of their inputs. Convolutional layers' neurons are organized into feature maps. Each neuron in a feature map has a receptive field that encodes a specific feature, and is linked to a neighborhood of neurons in the previous layer by a set of trainable weights known as a filter or kernel. The number of neurons in each hidden layer defines how many features will be represented at each layer. To compute the feature map, consider the kernel to be a sliding window that moves along the spatial dimensions of the input, evaluating a dot product element-wise among its entries and the input's selected entries. Different kernels generate different outputs (feature maps) from the same input. In the case of colored image (RGB), the kernel will have a different weight for each RGB channel and the output is computed by summing up the output of convolutions over each channel, look to figure 2.6 for more clarification. The hyperparameters of the convolutionl layer are kernel size, the stride which defines the step size by which the kernel move, and the padding that defines how many pixels (usually with zero value) will be added to surround the image to preserve image size in the feature map.

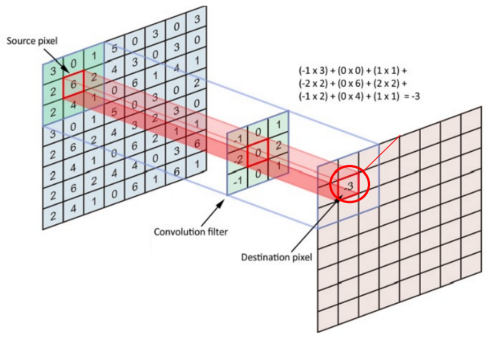


Figure 2.5: The $3x3$ kernel slides as a window over the image(source pixel) and the dot product is computed to produce feature map.

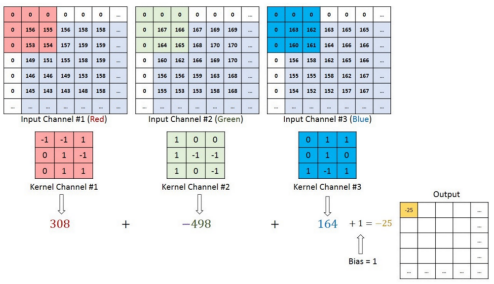


Figure 2.6: The input image is RGB with 3 channels, so there are three kernels with different weights, one for each channel. the output is computed by sum up the outputs from the convolution of each input channel with its kernel .

Pooling layers' purpose is to reduce the spatial resolution of the feature maps, achieving spatial invariance to input distortions and translations. As the kernel in convolutional layer, the slider window moves over the input but instead of computing a dot product, it outputs the maximum or the average value in the window depending on the type of pooling (average or max pooling). Pooling layer is characterized by window size, stride, and padding.

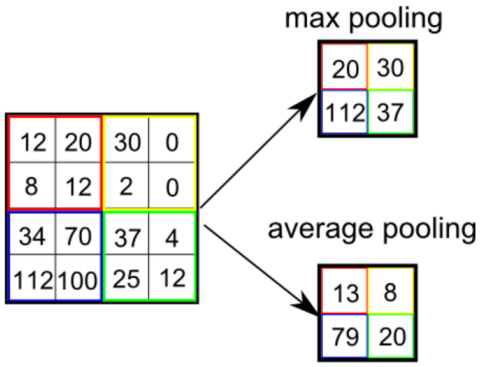


Figure 2.7: The max pooling returns the maximum value in the specific window , and average pooling returns the average of the values in the specific window.

CNNs can have one or more fully connected layers following convolutional and pooling layers. Convolutional and pooing layers extract abstract feature representation and Fully connected layer interpret these features and perform high-level functions like classification and prediction. In the classification case in the last fully connected layer, refer to as classification layer SoftMax activation function is used to get probabilities of the input being in a particular class instead of ReLU.

2.3.3 Backpropagation

Backpropagation is an optimization algorithm used to learn CNN parameters θ (weights w, bias b). It trains CNN by applying gradient descent to a cost function (empirical loss L_s). Backpropagation algorithm is composed of two passes, the forward and the backward pass. In the forward pass, the input propagates forward through the network layer by layer to produce the output, and during this pass, the network parameters don't change. The difference between the predicted output and the real output is calculated, and this error is propagated backward through the network (backward pass). The partial derivative of the error is computed backward with respect to all network parameters starting from the output layer using chain rules to know how much of the loss every node is responsible for. The parameters θ are updated at each time t toward the negative gradient using the updating rule:

$$\theta_k(t+1) = \theta_k(t) - \eta \cdot \frac{\partial l(t)}{\partial \theta_k(t)} \tag{2.9}$$

Loss function $l(t)$ is evaluated on random batch at step t, and learning rate η is a training hyperparameter that quantifies how much a model weight can change at every step.
Instead of computing the loss and updating the weights after passing all the training data (batch gradient descent), a better approach is employed in the batch gradient descent, where instead of using a single sample for the optimization (as in stochastic gradient descent), a group of them is used (batch) this allows attaining a much more stable estimate for the loss function gradient which, in turn, leads to better overall convergence.

2.4 ResNet-18

ResNet, short for a residual network is a deep convolutional neural network with residual block or skipped connections. In our work, we used ResNet as our model to perform feature extraction from input images.
Training a deep convolutional neural network using backpropagation is problematic due to gradient vanishing. The propagation of the gradient backward through multiple layers makes the gradient smaller and smaller leading to saturation or even degrading in the network performance. The use of residual blocks solves the vanishing gradient problem through the use of skip connections that leads to network compression at first and then exploration of multiple features of the input. The idea is that we skip the layers that are not useful for reducing the loss.

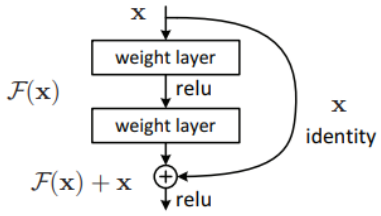


Figure 2.8: Residual Block.

- If the input x and the output y have the same dimension :

$$y = F(x) + x \tag{2.10}$$

- If the dimension is different , a linear projection W_s is used to match the dimension:

$$y = F(x) + W_s x \tag{2.11}$$

There is a variant architecture of ResNet, but we adopt ResNet-18 to have a comparable result with [7]. ResNet-18 is CNN, and the architecture is the same as ResNet-34 which is shown in figure 2.10, but there is a little difference are illustrated in the bellow table. The convolutional layers mostly have 33 filters and follow two simple design rules: (i) for the same output feature map size, the layers have the same number of filters, and (ii) if the feature map size is halved, the number of filters is doubled to preserve the time complexity per layer [10]. The down-sampling is directly applied by using convolution layers with a stride of 2. The last layers of the model are the average pooling layer and a 1000-way (in our case 100) fully connected layers with SoftMax activation function to perform the classification task.
The solid line in figure 2.10 represents identity mapping. When the dimensions increase (dotted line shortcuts) in figure 2.10, we consider two options:

- The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter.

- The projection shortcut is used to match dimensions (done by 11 convolutions).

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2.9: The first two columns of the table describe the difference between the architecture of the ResNet-18 and ResNet-32. Building blocks are shown in brackets, with the numbers of blocks stacked. Down sampling is performed by conv3-1, conv4-1, and conv5-1 with a stride of 2.

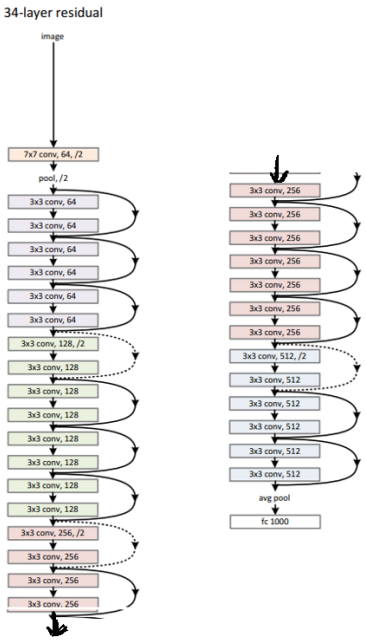


Figure 2.10: A residual network with 34 parameter layers ResNet-34.

2.5 Dataset

To train, test, and evaluate the performance of our model, we used the Cifar-100 dataset. Cifar-100 [11] has 100 classes, and they are grouped into 20 superclasses. Each image has a fine label representing the class to which it belongs and a coarse label determines the superclass to which it belongs. Cifar-100 has 500 training images and 100 testing images for each class. The images are colored images (RGB) and have a size of $32x32$.

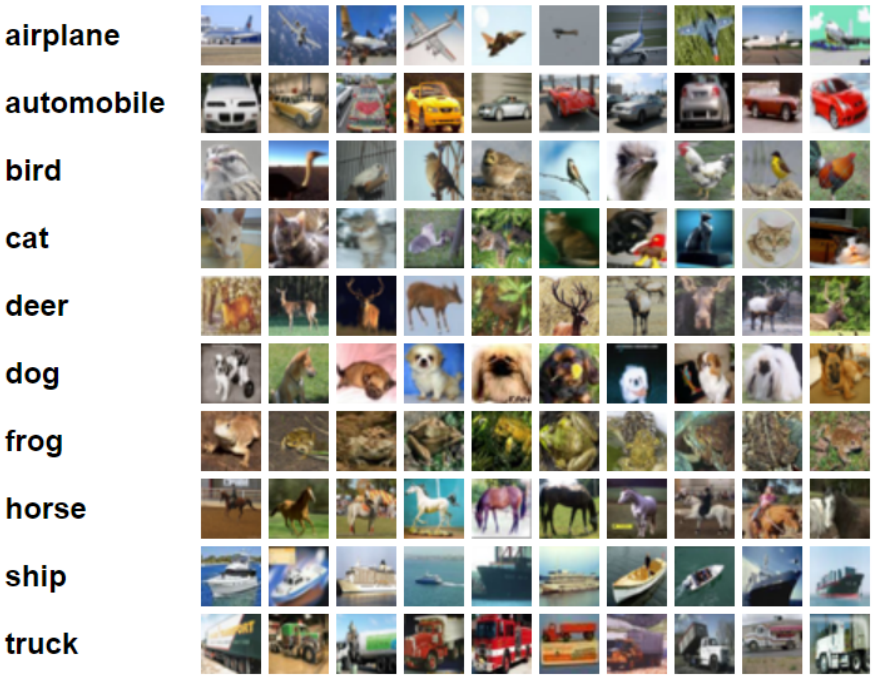


Figure 2.11: Cifar-100 dataset images samples.

3
Incremental learning

Deep Neural Network models typically require all training data to be available at once for training. However, in real-world scenarios, data may be received in small batches or incrementally over time. To address this, models need to be able to learn new data while preserving knowledge learned from previous data, a process known as incremental learning. In this chapter, we will discuss the concept of incremental learning, as well as techniques and evaluation metrics used in this type of learning.

3.1 Incremental learning

Incremental learning is the capability of machine learning architectures to continuously improve the learned model by feeding new data without losing previously learned knowledge [12]. This behaviour is inherently present in the human brain, which is continuously able to incorporate new data while preserving previously acquired Knowledge. Incremental learning has already been deployed in image classification and object detection [13, 14, 15]. One of the key benefits of incremental learning is that it allows the model to continuously adapt and improve over time, rather than being trained on a fixed dataset and then remaining unchanged. This can be particularly useful in applications such as online advertising, where the data is constantly evolving and the model needs to be able to adapt to changes in user behavior and preferences. Deep neural networks,though, tend to forget previously learned tasks when trained on a new task, a phenomenon known as catastrophic forgetting [3, 4, 5]. There are three scenarios for

incremental learning [1]:

- Task incremental learning: model architecture used in this scenario has a multi-headed output layer, meaning that each task has its own output units but the rest of the network is (potentially) shared between tasks [1]. The task boundary is known, and at inference time, task identity is known. The learned model can discriminate between classes belonging to the same task but not from different tasks.

- Domain incremental learning: the structure of the task remains the same, while the input distribution is changing. There are no task boundaries, and at test time task identity is not known.

- Class incremental learning: the task boundaries is known but the task identity is not known at inference time. here the learned model should discriminate between all classes learned so far.

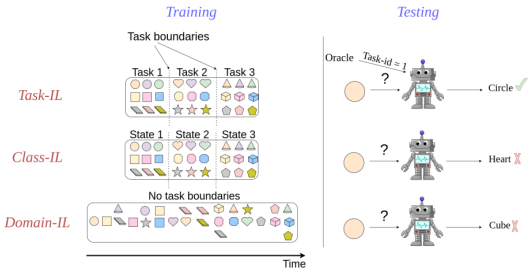


Figure 3.1: The three scenarios of incremental learning.

The following sections will introduce in more detail class incremental learning because, in our work, we are more interested in class incremental learning for image classification.

3.2 Class incremental learning (CIL)

Class incremental learning when a neural network model (learner) is trained on a series of tasks, and in each task it learns several classes disjoint from other classes in previous or future tasks. During the training process, learner has access only to classes belonging to the current task and should be able to classify all classes learned so far. The purpose of class incremental learning is to learn a unified model that can classify unseen data of all previously learned classes.

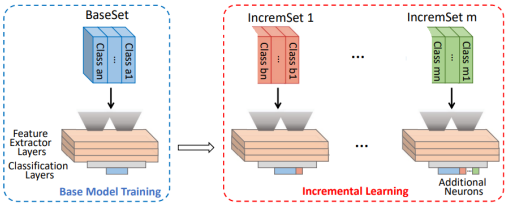


Figure 3.2: Illustration of class incremental learning scenario.

Practically, CIL problem is composed of n sequence tasks $\{(C^1, D^1), (C^2, D^2), ...(C^n, D^n)\}$. where each task t is represented by a set of classes $C^t = \{C^t_1, C^t_2, ...C^t_{n^t}\}$ and training data D^t, $D^t = \{X_t, Y_t\} = \{x_{t,j}, y_{t,j}\}_{j=1}^{N_t}$ is the training dataset of task t, N_t is the number of data, and $y_{t,j} \in C^t$ is a class label corresponding to an input $x_{t,j}$. During training a task t has access only to D^t and class set are disjoint between tasks ($C^{ij} = \Phi$ if $i \neq j$).

Our class incremental learning model consists of a feature extractor F_θ (ResNet-18) and a classifier G_φ (fully connected layer). At each step t, the goal is to minimize a loss function L on a new dataset D^t while also maintaining or improving upon the performance on previous tasks.

$$\theta_t, \varphi_t = argmin_{\theta_t, \varphi_t} L_t(G(F(X_t, \theta_t), \varphi_t), Y_t) \tag{3.1}$$

- θ_t, φ_t are the feaatures extractor (F) and classifier (G) parameters optimized in step t.
- $L_t(G(F(X_t, \theta_t), \varphi_t), Y_t)$ is the current model (at step t) loss on all previously learned dataset D^i and also on the current dataset D^t.

At each step $t \in [1, n]$ the current model receives only new data D^t and is required to learn those data without forgetting old data D^i $\forall i \in [1, t-1]$. The problem of balancing between maintaining knowledge from previous tasks and learning new information from current task is known as the stability-plasticity dilemma [6].

For L_t, a cross entropy loss is used, also known as the log loss, is a common loss function used in classification tasks. It measures the distance between the predicted probability distribution and the true probability distribution for a given class. The loss is calculated as the negative logarithm of the predicted probability for the correct class. It is a widely used loss function because it is easy to compute, and has the property that it penalizes large errors more than smaller ones. The cross entropy loss is defined as:

$$L_{ce} = -\sum_i y_i \log(p_i) \tag{3.2}$$

where y_i is the true label for class i and p_i is the predicted probability for class i. The loss is minimized when the predicted probabilities are close to the true labels.
Two main challenges exist in class incremental learning (CIL): classifier bias and task-level overfitting. Classifier bias can occur when the decision boundary learned from the initial dataset is dramatically changed by the incorporation of new data, leading to an unbalanced treatment of the different classes. Task-level overfitting occurs when the model becomes too specific to the training data and is unable to generalize to new data. This can be difficult to avoid in CIL, as the model needs to continuously adapt to new data and integrate it into the model in a way that avoids overfitting. To address these challenges in CIL, it is important to focus on learning task-agnostic representations, which can help to improve the model's generalization ability and reduce bias.
There are several approaches that have been developed to address the challenges of class incremental learning. These approaches can be divided into three categories: regularization-based methods, rehearsal based methods, and bias-correction methods.

- Rehearsal based methods [16, 17, 18] : these strategies involve storing a portion of the old data and using it to jointly train the model with the current data. However, These approaches is not always feasible due to memory limitations or privacy concerns. As an alternative, some researchers have suggested using deep generative models to generate pseudo-samples of previous classes. While this approach can be effective in some cases, it can be inefficient for complex datasets and is also prone to catastrophic forgetting.

- Regularization methods [19, 20, 21]: involve adding a penalty term to the classification loss function of the original model in order to regularize the important parameters and prevent them from being changed during the training of a new task. This approach is suitable when multi head classifiers or task incremental learning are used only in case task identifier, an alternative solution is to use knowledge distillation to perform regularization.

- Bias-correction methods [22, 23, 24]: aim to equalize the performance of the model on classes from all tasks, by adjusting the network's bias or classifier norm. This bias is often caused by the fact that a model tends to have an advantage for classes that have been recently learned, leading to a larger classifier norm and a bias towards these classes. As an example of this approach is adjusting the norm of new class weight vectors to those of the old class weight vectors.

In this work, we adopted knowledge distillation technique, and a novel method called PASS developed in [7]. PASS consists of prototype augmentation, where one class-representative prototype is created for each old class in the deep feature space to maintain the decision boundary of

previous tasks, and self-supervised learning (SSL) to learn more generalizable and transferable features for other tasks. We will elaborate on these techniques in more detail in the following subsection.

3.2.1 Prototype Augmentation

The main idea behind prototype augmentation is to maintain the decision boundary of previous tasks by memorizing one class-representative prototype for each old class and augmenting the deep feature space with these prototypes during training on new tasks.
To implement prototype augmentation in an incremental learning setting, the model is first trained on the first task and the class-representative prototypes are computed for each class in the task. These prototypes are then added to the deep feature space, which allows the model to maintain the decision boundary for the classes in the first task while learning the classes in the second task. This process is repeated for each subsequent task, with the class representative prototypes for all previous tasks being added to the feature space.
Instead of store sample from previous task, one prototype for each old class is computed and added to the deep feature space. The class prototype is obtained by taking the mean of all the features of the images belonging to that class as shown in 3.3.

$$\mu_{t,k} = \frac{1}{N_{t,k}} \sum_{n=1}^{N_{t,k}} F(X_{t,k}, \theta_t) \tag{3.3}$$

When learning new task, the prototype of each old class, e.g. class k_{old} at stage t_{old}, is augmented as below:

$$F_{k_{old},t_{old}} = \mu k_{old}, t_{old} + e * r \tag{3.4}$$

$$r^2 = \frac{1}{K_1 * D} \sum_{k=1}^{K_1} Tr(\Sigma_{1,k}) \tag{3.5}$$

- e is a Gaussian Noise $N(0, 1)$
- r is a scale to control uncertainty of augmented prototype
- K_1 is the number of classes in the first task
- D is the dimension of the deep feature space
- $\Sigma_{1,k}$ is the covariance matrix for the features of class k at first stage

During training session, at each step t, only the current data set D^t is available. After the model has learned the current classes, a prototype for each of these classes is computed using equation 3.3. In the next step $t + 1$, the prototypes for each old class are augmented using equation 3.4 and fed to the classifier along with the deep features of new task data for classification. This helps to maintain the discrimination and balance between old and new classes when learning a new task (see figure 3.5). The prototypes are augmented via guessing noise to allow the classifier to account for the uncertainty and variability in the features of old classes.

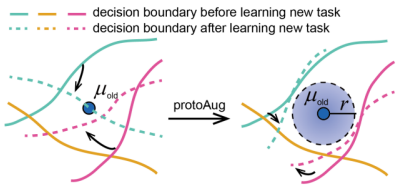


Figure 3.3: prototypes augmentation helps to maintain the discrimination and balance between old and new classes while learning a new task, thus preventing the decision boundary from being dramatically changed and preventing catastrophic forgetting [7].

3.2.2 Self supervised learning based label augmentation

Self supervised learning based label augmentation combines self supervised learning with label augmentation to train the model to predict the transformations applied to the input data, rather than relying on explicit labels. This helps the model learn richer features.
Label augmentation is a technique used to increase the number of training examples available for a machine learning model by generating additional examples from the existing ones. This is typically done by applying various transformations to the input data, such as rotating, flipping, cropping, or adding noise, and assigning new labels to the transformed examples. The goal of label augmentation is to improve the generalization performance of the model by providing it with more diverse training data, which can help it learn more robust and invariant features. Here rotation transformation is used, the training data for each class is rotated by 90, 180, and 270 degrees to create 3 new classes. This extends the original K-class problem to a new $4K$-class problem, with the goal of helping the model learn more robust and generalizable features. The rotated data is assigned new labels using self supervised learning to be used for training.

$$X_t^{'} = rotate(X_t, \theta) \ , \theta \in \{90, 180, 270\} \tag{3.6}$$

Self supervised learning (SSL) refers to learning models with the use of automatically generated labels. Compared to supervised learning methods which require a data pair X_i and Y_i while Y_i is annotated by human labor, self-supervised learning is also trained with data X_i along with its pseudo label P_i while P_i is automatically generated for a predefined pretext task without involving any human annotation [25].

New labels $Y_t^{'}$ are assigned to augmented samples $X_t^{'}$ used 4-way self supervised tasks(is a type of SSL), for more information refer to [26].

The above approach of using self supervised tasks to augment labels during training can help to relax certain invariant constraints, allowing the model to learn richer and more generalizable features by learning both the original task and the self-supervised tasks simultaneously.

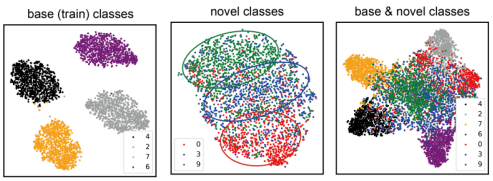


Figure 3.4: Deep features space for base and novel classes without SSL.

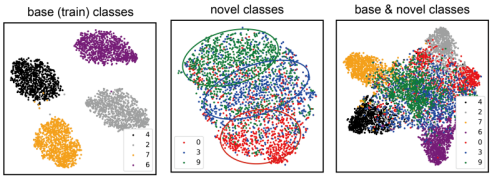


Figure 3.5: SSL helps to better distinguish the distribution of novel classes, resulting in a decrease in overlap between previously learned (base) classes and new classes. [7]

3.2.3 Knowledge distillation

Knowledge distillation (KD) refers to the process of transferring knowledge from a pretrained model (called the "teacher" model) to a new model (called the "student" model) that is being trained to perform the same or a similar task. The goal of knowledge distillation is to improve the performance of the student model, by leveraging the knowledge and expertise of the teacher model.
There are several ways to perform knowledge distillation, but one common method is to train the student model to imitate the teacher's output. This is done by minimizing the difference between the teacher's output and the student's output, using a distance measure such as mean mean-squared error or the Kullback-Leibler divergence.
In class incremental learning the teacher model is a model that has been trained on all tasks up to the current one, and the student model is a model that is being trained on the current task. In other words, the knowledge of previous model F_{t-1} is transferred to the new model F_t as model F_t learns a new task, with the goal of helping model F_t learn the new task while still preserving the knowledge from F_{t-1}. By transferring the knowledge of model F_{t-1} to model F_t, model F_t can retain a better representation of the previous tasks, which can help to mitigate the effects of catastrophic forgetting.

3.3 Evaluation metrics

There are several evaluation metrics that are commonly used to assess the performance of a class incremental learning (CIL) model. Here we use accuracy and average forgetting as evaluation metrics.

3.3.1 Accuracy

Accuracy is defined as the proportion of correctly classified samples in the total number of samples. It is calculated by dividing the number of correctly classified samples by the total number of samples, and then multiplying by 100 to get the percentage accuracy.
In class incremental learning, accuracy is typically computed as the average accuracy of all the classes that have already been learned. This allows us to assess the model's overall performance on all tasks, rather than just a single task.

3.3.2 Average forgetting

Average forgetting is defined as a measure of how much the performance of the model on previous tasks has decreased after learning new tasks. It is calculated as the average of the percentage change in performance on all previous tasks after learning a new task. A lower average forgetting value indicates better performance in retaining knowledge of previous tasks while learning new tasks. Average forgetting is an important evaluation metric in CIL because it helps to identify how much the model is prone to forgetting previously learned knowledge and how well it can retain this knowledge while learning new tasks.
The average forgetting measure F_k is calculated by averaging the forgetting measure for all tasks i that were learned before task k.

$$\begin{aligned} F_k &= \frac{1}{k-1}\sum_{i=1}^{k-1} f_k^i. \\ f_k^i &= max_{\{t\in 1,\dots,k-1\}}(a_{t,i} - a_{k,i}) \quad \forall i < k \end{aligned} \tag{3.7}$$

- f_k^i is the forgetting measure of the i-th task after training k-th task.
- $a_{k,i}$ is the accuracy of task i after training task k.

4

Models

This thesis builds on the work proposed in [7], which have been further discussed in chapter 3. Based on this framework, we developed some extensions to improve the performance of the proposed model, with a focus on analyzing its performance under different class ordering and incremental scenarios. Specifically, we have examined the impact of class order on class incremental learning (CIL).

The base model for class incremental learning was developed using PyTorch [27] implementation of ResNet-18 as a starting point. The ResNet-18 (F) extracts features from RGB input images and is followed by a fully connected layer (classifier) G that classifies both the base and new classes incrementally.
The training data sets are augmented using rotation-based transformations outlined in subsection 3.2.2, before being applied to the feature extractor to learn more generalized and richer features.

In the subsequent section, we will present the loss function used to train our model, and perform an analysis of the impact each term of the loss function has on the feature space. Mathematical definitions of each term will be provided. Separate sections will be dedicated to the discussion of class ordering and the various incremental scenarios adopted during the training of our model.

4.1 Loss function

The model (ResNet-18+classifier) is trained to minimize a loss function on the training dataset D_t to learn new classes without forgetting previously acquired knowledge. The loss function as outlined in [7] is composed of cross entropy loss L_{ce}, prototype augmentation loss $L_{protoAug}$, knowledge distillation loss L_{kd} and we add a novel loss which is clustering loss L_{clus}. The loss function at each incremental step t can be defined as:

$$L_t = L_{t,ce} + \alpha L_{t,protoAug} + \beta L_{t,kd} + \gamma L_{t,clus} \tag{4.1}$$

- α, β, γ are hyperparameters and represent the weighting factors for the losses

4.1.1 Cross entropy loss

Cross entropy loss L_{ce} is commonly used in multi-class classification tasks to train a model to predict the correct class for each input sample. L_{ce} is calculated for each sample in the training batch and then the values are averaged across all samples to obtain the final loss value.

$$L_{ce} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log(p_{i,j}) \tag{4.2}$$

Where,

- N is the number of samples in the batch
- C is the number of classes
- $y_{i,j}$ is the true label of class j for sample i, where $y_{i,j} = 1$ if sample i belongs to class j, and 0 otherwise
- $p_{i,j}$ is the predicted probability of class j for sample i

$p_{i,j}$ is computed by applying the softmax function to the output of the last layer of the model, as previously mentioned in chapter 2.

4.1.2 Knowledge distillation loss

The purpose of knowledge distillation is to preserve the knowledge gained through incremental learning. As discussed in subsection 3.2.3. The previously learned model will be used at each training step to guide the preservation of previous classes. We don't store any previous images or labels from training data, we store only the previous models. The fundamental concept behind knowledge distillation is to make the current model, F_t, replicate the output of the previous model, F_{t-1}, so that the model parameters, θ_t, can recognize the previously learned classes using θ_{t-1}.
Practically, at training step t, we load the previous model F_{t-1} and we apply input x_t from the current training dataset batch to both models. Then, we calculate the difference between the output of the two models by using the L2 norm in the feature space, as represented in equation 4.3. By minimizing this difference, we enforce the current model's output to be similar to the previous model's output, thus preserving the previous knowledge.

$$L_{t,kd} = \|F_t(x_t, \theta_t) - F_{t-1}(x_t, \theta_{t-1})\| \tag{4.3}$$

4.1.3 Prototype augmentation loss

As previously stated, our method is non-exemplar, meaning that we do not store any specific samples of old classes. Instead, we create a class-representative prototype in the deep feature space to remember each class. When learning new tasks, these old prototypes are augmented with disturbances as defined in equation 3.4 and passed to the unified classifier for classification. At training step t, prototype augmentation loss $L_{protoAug}$ is calculated as follow

$$L_{t,protoAug} = \sum_{i=1}^{t-1} L(G(F_i, \varphi_t), Y_i) \tag{4.4}$$

where F_i is the feature augmentation of old classes C_i.
The Augmentation of old class prototypes to the feature space will help to maintain the distribution of old classes (Figure 4.1) while learning new classes, thereby reducing catastrophic forgetting phenomena.

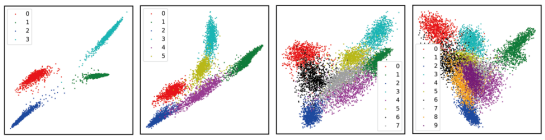


Figure 4.1: The first image depicts the distribution of the base classes in the feature space. In the subsequent figure, three classes were incrementally added, and it is obvious that the distribution of the original classes is still preserved up to a certain point

4.1.4 Clustering loss

The clustering loss term is derived from the Clustering approach, which is a popular unsupervised learning technique used for grouping similar objects into classes or clusters. it identifies patterns in the data and groups similar observations or items based on their characteristics or features. In the context of image classification, clustering can be used to extract features from images and then group those features based on their similarity. Distance functions such as Euclidean distance or cosine similarity are used to measure similarity between data, and clustering minimizes the distance function to group similar data into one cluster.
The Clustering loss objective is to tighten feature vectors within a class around the centroids and exert a repulsive force on the centroids of different classes, causing them to move apart.

Given a batch of training data $D_i = [x_i, y_i]$ with N samples, we extract features $F(X_i)$ and compute the centroids for each class by averaging the corresponding features. Then, the clustering loss is calculated as follows:

$$L_{t,clus} = \frac{1}{N}\sum_{i=1}^{N} d(F(x_i), c_{y_i}) - \frac{1}{C}\sum_{i\in C}\sum_{\substack{j\in C \\ j\neq i}} d(c_i, c_j) \tag{4.5}$$

where,

- $F(x_i)$ is the extracted feature of the input image x_i
- c_{y_i} is the centroid corresponding to the label y of the i-th feature.
- C is the number of unique labels seen until now (previous label) including distinct labels in the current batch.
- d() represents a generic function to measure the distance.

The first component measures the proximity of the feature vectors to their respective centroids, while the second component evaluates the distance between the class centroids. For the distance function, we employed the $L1$ and $L2$ norms, with better results obtained using the $L1$ norm. The centroids vector also includes a unique label from previous tasks (prototypes). The aim of minimizing the clustering loss $L_{t,clus}$ is to center the feature vectors around their respective centroids and to increase the separation between the clusters corresponding to different classes. This reduction of confusion between classes will help the model effectively identify and distinguish between different classes while retaining its knowledge of previously learned classes.

4.2 Class Incremental learning scenarios

Our objective is to train a model incrementally, where the training dataset is divided into several partitions and the training process is executed in a sequence of steps, with the model only having access to one partition at each stage. The partitions are disjoint. To assess the performance of the model, we perform experiments on different incremental setups to investigate its ability to handle both simple and complex incremental learning tasks. The Cifar-100 dataset is used as the benchmark for our evaluation, as outlined in section 2.5. To define an incremental learning setup, we need to specify the following:

- The set of learnable classes C, in our case we have $|C| = 100$
- The set of tasks s, which is represented as $s = s_1, \ldots, s_N$, where N is the number of incremental steps.
- The set of classes C_k, to be learned at each step s_k, which is a subset of C. It is required that the union of all C_k overall steps should equal to C, and the sets of classes C_j and C_k must not have any overlapping elements for $j \neq k$.
- The training dataset D_k for each task s_k and it only contains input samples that belong to the class set c_k.

The training procedure for our model begins with a set of base classes, and at each subsequent training stage s_k, an additional number of classes are introduced. The specific incremental learning scenarios that our model is trained on are defined in the below table.

Incremental steps	Base classes	Number of added classes per step
1	50	50
4	20	20
5	50	10
9	10	10
10	50	5
19	5	5
20	40	3

Table 4.1: CIL training scenarios

4.3 Class ordering

Model performance and ability to learn and generalize to new classes can be affected by the order in which classes are introduced to the model. To study the effect of class order, we defined different class ordering strategies for Cifar-100 dataset and evaluated our model's performance on these different strategies. The class ordering strategies we used are:

- Random order: permute the original class order of Cifar- 100 dataset, which is an alphabetic order. Random ordering can provide a baseline for comparing the performance of the model with other methods of class ordering since it accounts for the variability in performance due to the order of classes. Refer to AppendixA for the original order of the CIFAR-100.
- Order Cifar-100 based on a predefined grouping or taxonomy. The original Cifar-100 dataset is divided into 20 groups or superclasses, each of which contains 5 macro classes (look to Table 7.1 in AppendixA).
  In this ordering approach, the new classes provided to the model in each incremental step are ordered based on the superclasses, meaning that they belong to the same group. This approach helps to evaluate the model's ability to learn related classes together, and its ability to handle the complexity of the incremental learning task.

In the upcoming chapter, we will present the outcomes of the proposed model under various training scenarios based on both random and superclass class orders.These results will examine the performance of our model under different conditions and provide a comprehensive evaluation of its effectiveness.

5 Results

In this chapter, our objective is to evaluate the proposed framework in chapter 4. To achieve this, we consider two settings for the model. The first setting involves training the model using only the cross-entropy loss, distillation loss, and prototype augmentation loss, while in the second set, we add the clustering loss to these terms. We trained the model using different incremental steps to assess the impact of scenario complexity on the model's performance. Additionally, to analyze the effect of class order on the model, we employed two distinct class orders for the dataset: random order and superclass order.
In section 5.1, we will provide a description of the implementation and training details, in the following sections, we will report the evaluation of the model on the test dataset for both settings and compare their respective performances.

5.1 Implementation Details

The framework was implemented using Pytorch, and Tensorboard was used for monitoring the learning process and visualizing the model output. The model was trained on a single Nvidia Geforce GTX 1070 GPU, which had 8GB of dedicated memory. Furthermore, the most extended training session conducted with this setup took slightly more than 24 hours to complete.

In our experiments, we utilized ResNet-18 as the base model and added a fully connected layer

on top of it to construct a unified classifier. The model was initially trained on the base classes, after which we sequentially incorporated the remaining classes. We evaluated our approach using different incremental learning scenarios, including starting with 50 classes and adding the remaining 50 classes in the final step, starting with 20 classes and adding 20 classes in each of the following four steps, and additional scenarios with 5, 9, 10, 19, and 20 phases. After each phase, we assessed the model's performance in all the previously learned classes up to that point.

We trained the model using Adam optimizer [28] with an initial learning rate of 0.001 and a batch size of 64. We continued training the models for 100 epochs, and at the 45th and 90th epochs, we reduced the learning rate by multiplying it by 0.1.
We trained and tested our model on the CIFAR-100 dataset. The training dataset contains 5000 samples, with 500 images per class, while the testing dataset composes of 1000 samples, with 100 images per class, both of size 32×32. Before training each batch, we augmented images in the batch by rotating them by 90°, 120°, and 270°, and hence the classification problem was extended from k class to $4K$ class by adding 3 new classes. We utilized self supervised learning to assign labels for the newly generated images.

5.2 The loss function excluding cluster loss

In this section, we will present and analyze the results of optimizing the loss function expressed in Equation 4.1 while excluding the cluster loss term. We set the weight balances for prototype augmentation loss and knowledge distillation loss, namely α and β, to 10, as proposed in [7].

In Table 5.1, we compare the classification accuracy results of our Class Incremental Learning (CIL) method on Cifar-100 dataset using two distinct class orders for different incremental learning scenarios. Additionally, Figure 5.1 illustrates the accuracy of the current model at each incremental step on all previously learned classes, including the current classes.

For the random order strategy, The model achieved an accuracy of 63.84% in a one-step task, starting with 50 base classes and 50 added classes. However, the accuracy decreases as the number of base and added classes decreases in subsequent tasks. For example, in 9 tasks, which include only 10 base classes and add 10 classes at each training session, the accuracy is reduced to 35.26%. The worst performance is observed in 19 incremental steps, which add 5 classes at each step, resulting in an accuracy of only 21.07%.

On the other hand, when adding classes in the superclass order, the performance of the model is generally lower than that of the random order strategy. For the first task, which included one incremental step, the accuracy is 61.79%, which is lower than the corresponding accuracy of 63.84% obtained with the random order strategy. The worst performances are observed in 9 and 19 sequential tasks with an accuracy of 30.67% and 13.6%, respectively.
We expand our analysis by evaluating our model also on another class order setting, where we initially train the model on 20 classes as base classes, one from each superclass. We then add the remaining classes on the same basis (one class from each superclass), in the following four steps. The performance of this setting can be seen in Figure 5.1(h). The Accuracy is 47.65% which is the same as in random order 47.29% for 4 incremental steps.

Table 5.2 displays the forgetting results of our incremental model at the last incremental step on all previously learned classes. The results show that the forgetting rate is 16.14% for a single incremental step, and it increases to 28.55% after 20 incremental steps. This suggests that the forgetting phenomenon becomes more severe as the number of sequential tasks increases. Please refer to Figure 7.11, and Figure 7.12 in AppendixA for the detailed forgetting rates at each incremental step.

The lower performance of the superclass order strategy can be attributed to the grouping of classes from the same superclass, which results in the increased similarity between them. This makes it more challenging for the model to learn discriminative features and leads to easier forgetting over time. For instance, in 19 tasks, where each step adds a superclass (5 macroclasses), the accuracy is the lowest at 12.15%. It is worth noting that in this case also, the classes at each incremental step are different from each other because they are from different superclasses (for example, at step $t-1$, we add the fish superclass and at step t, we add the flower class), which makes the learned features more prone to modification over time.
Moreover, it is clear that when starting with a large number of base classes, even if the number of sequence tasks is big, the performance does not decrease significantly as in the cases of 19 and 9. This is because the model is exposed to different classes at the first training stage and can learn to discriminate between them effectively. A larger number of classes generally results in better performance, most likely because anchoring to the first task already yields a more diverse set of features.

In conclusion, the complexity of incremental steps significantly impacts the model's performance, while the class ordering strategy has no significant effect on the accuracy of the model.

Incremental steps	**Base classes**	**Added classes per step**	**Accuracy**	
			Random (Alphabetically order)	Superclass order
1	50	50	63.84	61.79
4	20	20	47.29	42.94
5	50	10	56.34	52.8
9	10	10	35.26	30.67
10	50	5	49.69	46.9
19	5	5	21.07	13.6
20	40	3	47.57	43.55

Table 5.1: CIL accuracy result on cifar-100 for different tasks

Phases	**Base classes**	**Added classes per step**	**Forgetting**	
			Random (Alphabetically order)	Superclass order
1	50	50	16.14	12.32
4	20	20	20.012	15.07
5	50	10	20.11	17.69
9	10	10	20.58	18.42
10	50	5	27.7	24.36
19	5	5	16.01	12.15
20	40	3	28.55	26.56

Table 5.2: CIL forgetting result on Cifar-100 for different incremental tasks

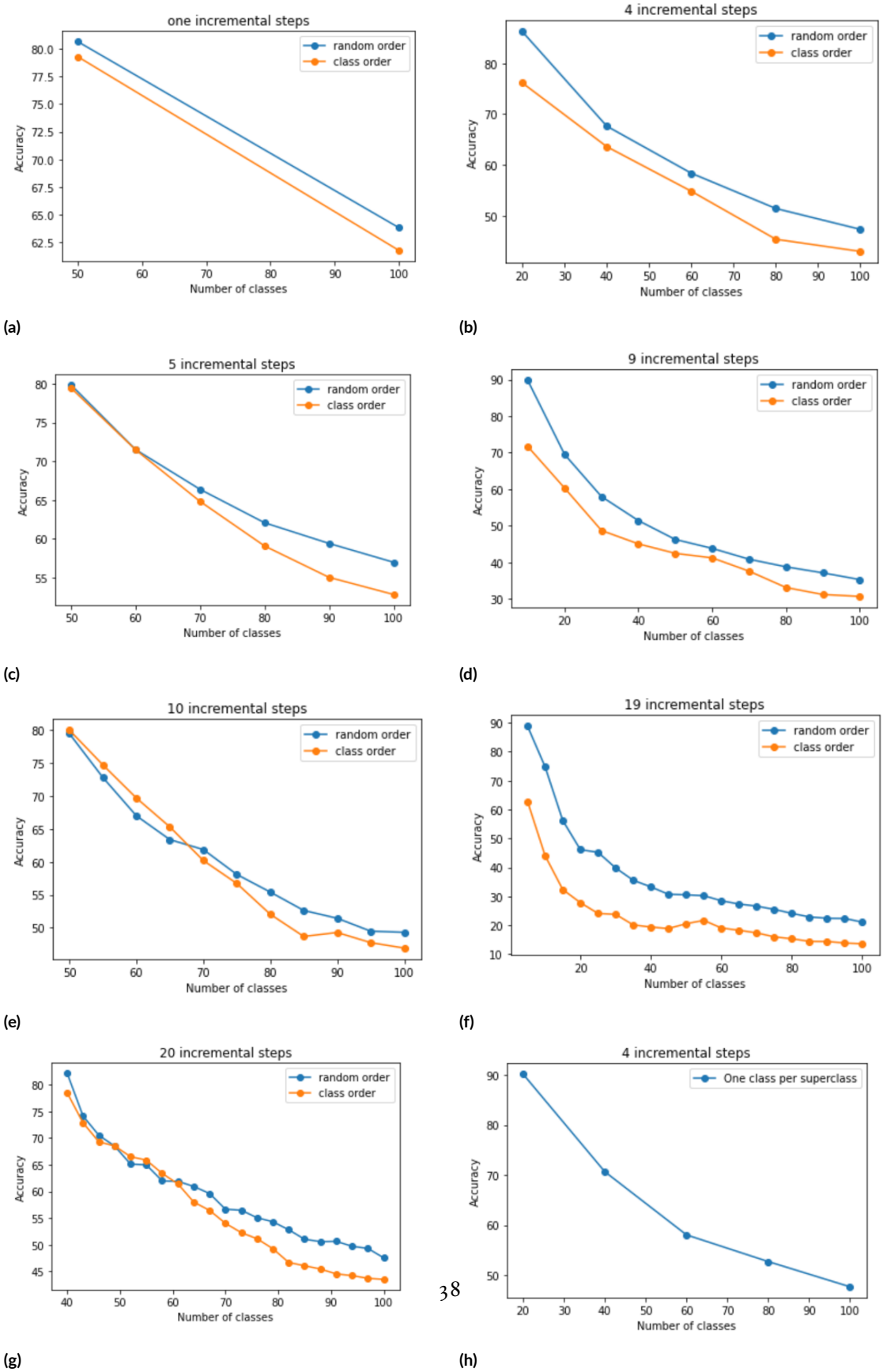


Figure 5.1: Results of classification accuracy on Cifar-100, which contains 1, 4, 5, 9, 10, 19 and 20 sequential tasks using random and superclass order

5.3 The loss function including cluster loss

The model was trained on the loss function 4.1, which included the cluster loss term, and the hyperparameter for the cluster loss was optimized over 10 sequential tasks. The results of the optimization are presented in Table 5.3, and it was found that the best performance was achieved with a hyperparameter value of $\gamma = 0.01$.
The classification accuracy on different sequential tasks (1, 5, 10, 19, 20) with $\gamma = 0.01$ is presented in Table 5.4. Surprisingly, it was found that the accuracy decreased for both random and superclass order when the cluster term was added. This indicates that the addition of the cluster term did not enhance the model's performance, but rather it had an adverse effect on the model's accuracy.

gamma	Tasks	Accuracy
10	10	32.77
1	10	30.85
0.1	10	28.37
0.01	10	33.83
0.001	10	32
0	10	49.69

Table 5.3: Hyperparameter selection for the cluster loss term

Incremental steps	Base class	Added classes per step	Accuracy	
			Random order	Superclass order
1	50	50	62.62	61.72
5	50	10	45.76	45.92
10	50	5	33.83	33.67
19	5	5	13.84	12.82
20	40	3	27.01	24.23

Table 5.4: CIL accuracy result with the use of cluster loss

6
Conclusion

This thesis aimed to investigate the impact of class ordering and task complexity on class incremental learning using the Cifar-100 dataset. To achieve this, we employed the model proposed in [7], which is a non-exemplar method that does not store examples from previous classes. Instead, it augments a prototype from each previously learned class in the feature space and uses self-supervised learning to learn rich and generalizable features. We considered two different class ordering methods, namely random order and superclass order, and evaluated the model's performance on different incremental learning scenarios consisting of 1, 4, 5, 9, 10, 19, and 20 tasks.

Our results indicate that the proposed model with random order performed slightly better than the superclass order, with a small difference in performance (around 5%), except for the case of 19 sequential tasks, where the difference was slightly larger (about 7%). Additionally, we observed that the model's performance decreased as the complexity of the incremental task increased. Specifically, the model's highest performance was achieved when we had only one incremental step (63.84%, and 61.79% for random and superclass order respectively), while the worst performance was observed in the case of 19 tasks, with 21.07% and 13.6% for random and superclass order, respectively. we can conclude that the performance of the model is not significantly affected by the order of classes, but the complexity of incremental tasks has a significant impact on the model's performance. The results also indicate that starting with a larger number of classes typically leads to superior performance. This is likely because anchor-

ing to the first task results in a richer set of features.

Furthermore, we proposed the use of a cluster loss to organize more the feature space and reduce confusion between new and previous classes. However, our experiments showed that this approach did not improve the model's performance, instead, it led to a decrease in the classification accuracy.

For future work, we suggest exploring different class ordering methods to further check the model's robustness. For instance, we could use confusion matrices to group highly miss classified classes together or group rarely miss classified classes together to feed them to the model in the same task. Additionally, we could test and train the model on other dataset to evaluate its performance.

References

[1] G. M. de Ven and A. S.Tolias, "Three scenarios for continual learning," *arXiv:1904.07734*, 2019.

[2] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, 1999.

[3] D. A. I.J.Goodfellow, M.Mirza and Y.Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *in Proc. Int. Conf. Learn. Repres*, 2014.

[4] N. J. G. A. K. J. T. J.Kirkpatrick, R.Pascanu and A.Grabska-Barwinska, "Overcoming catastrophic forgetting in neural networks," *National Academy of Sciences*, 2017.

[5] M.McCloskey and N.J.Cohen, "Catastrophic interference in connectionist networks:the sequential learning problem," *in Psychology of learning and motivation*, 1989.

[6] A. M.Mermillod and P.Bonin, "The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects," *Frontiers in psychology*, 2013.

[7] C. W. F. Y. Fei Zhu, Xu-Yao Zhang and C.-L. Liu1, "Prototype augmentation and self-supervision for incremental learning," *Proceedings of the IEEE*, 2021.

[8] S. P. Ashley Walker Robert Fisher and E. Wolfart, *Hypermedia Image Processing Reference*, online book ed., 2000.

[9] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Prentice Hall PTR, 1998.

[10] S. R. Kaiming He, Xiangyu Zhang and J. Sun, *Deep residual learning for image recognition*. In CVPR, 2016, p. 770–778.

[11] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*. Technical report, 2009.

[12] U. Michieli and P. Zanuttigh, "Incremental learning techniques for semantic segmentation," *in Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019.

[13] C. K.Shmelkov and K.Alahari, "Incremental learning of object detectors without catastrophic forgetting," *in Proceedings of International Conference on Computer Vision (ICCV),PP.3400–3409*, 2017.

[14] L. W. Y. Y. Z. L. Y. G. Z. Z. Y. Wu, Y. Chen and Y. Fu, "Incremental classifier learning with generative adversarial networks," *arXiv preprint arXiv:1802.00853*.

[15] G. S.-A.Rebuffi, A.Kolesnikov and C. Lampert, "Incremental classifier and representation learning," *in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR),PP.2001–2010*, 2017.

[16] G. S. ylvestre Alvise Rebuffi, Alexander Kolesnikov and C. H. Lampert, "icarl: Incremental classifier and representation learning," *In CVPR*, 2017.

[17] T. K. P. J. Oleksiy Ostapenko, Mihai Puscas and M. Nabi, "Learning to remember: A synaptic plasticity driven framework for continual learning," *In CVPR*, 2019.

[18] B. G. Rahaf Aljundi, Min Lin and Y. Bengio, "Gradient based sample selection for online continual learning," *In NeurIPS*, 2019.

[19] N. R. J. V. G. D. A. A. R. K. M. J. Q. T. R. A. G. B. e. a. James Kirkpatrick, Razvan Pascanu, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, 2017.

[20] M. E. M. R. Rahaf Aljundi, Francesca Babiloni and T. Tuytelaars, "Memory aware synapses: Learning what (not) to forget," *In ECCV*, 2018.

[21] B. P. Friedemann Zenke and S. Ganguli, "Continual learning through synaptic intelligence," *In ICML*, 2017.

[22] L. W. Y. Y. Z. L. Y. G. Y. Wu, Y. Chen and Y. Fu, "Large scale incremental learning," *in Proc. IEEE Conf. Computer Vision, Pattern Recognition*.

[23] N. G. C. S. F. M. Castro, M. J. Mar´ın-Jimenez and K. Alahari, "End-to-end incremental learning," *in Proc. Eur. Conf. Computer Vision*.

[24] E. Belouadah and A. Popescu, "Il2m: class incremental learning with dual memory," *in Proc. IEEE Int. Conf. Computer Vision*, 2019.

[25] L. Jing and Y. Tian, "Self-supervised visual feature learning with deep neural networks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell*, 2020.

[26] S. J. H. nkook Lee and J. Shin, "Self-supervised label augmentation via input transformations," *In ICML*, 2020.

[27] F. A. J. G.-T. Z. N. L. e. a. A.Paszke, S.Gross, "Pytorch: An imperative style, high performance deep learning library," advances in neural information processing systems," *Advances in neural information processing systems,vol. 32, pp. 8026–8037*, 2019.

[28] D. P. Kingma and J. Ba, "A method for stochastic optimization," *In ICLR*, 2015.

7

AppendixA

Cifar-100 **class order**:
In this section we present the two class ordering , which are used to train and test our model.

Cifar-100 dataset original order:
apple, aquarium-fish, baby, bear, beaver, bed, bee, beetle, bicycle, bottle, bowl, boy, bridge, bus, butterfly, camel, can, castle, caterpillar, cattle, chair, chimpanzee, clock, cloud, cockroach, couch, crab, crocodile, cup, dinosaur, dolphin, elephant, flatfish, forest, fox, girl, hamster, house, kangaroo, keyboard, lamp, lawn-mower, leopard, lion, lizard, lobster, man, maple-tree, motorcycle, mountain, mouse, mushroom, oak-tree, orange, orchid, otter,palm-tree, pear, pickup-truck, pine-tree, plain, plate, poppy, porcupine, possum, rabbit, raccoon, ray, road, rocket, rose, sea, seal, shark, shrew, skunk, skyscraper, snail, snake, spider, squirrel, streetcar, sun-flower, sweet-pepper, table, tank, telephone, television, tiger,tractor, train, trout, tulip, turtle, wardrobe, whale, willow-tree, wolf, woman, worm.

Superclass	Macroclass
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Table 7.1: Cifare-100 datatset superclasses and macroclasses

Confusion matrix:

Figure 7.1 to Figure 7.8 show the confusion matrices for different sequential tasks (1, 5, 9, 19) for random order and superclass order.

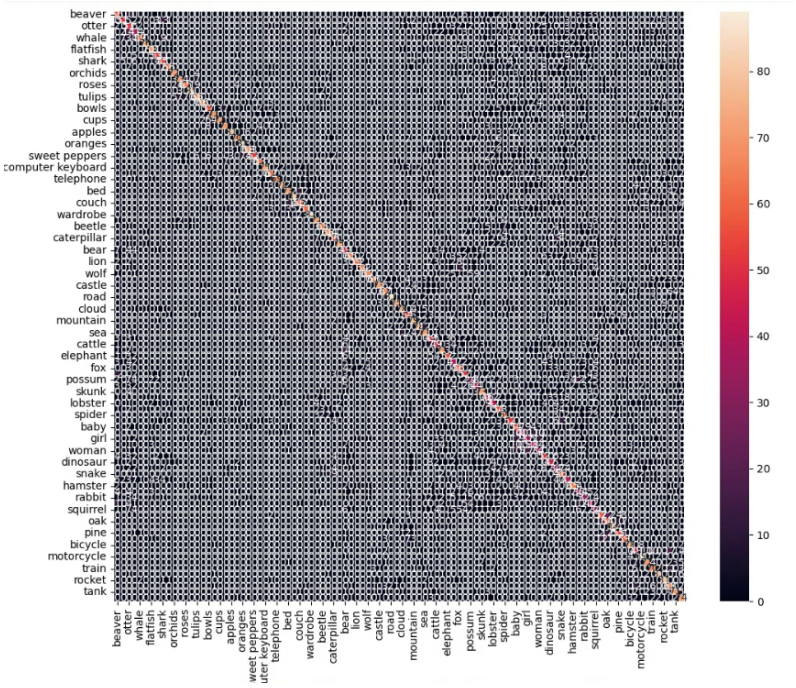


Figure 7.1: Confusion matrix for 1 sequential tasks for superclass order.

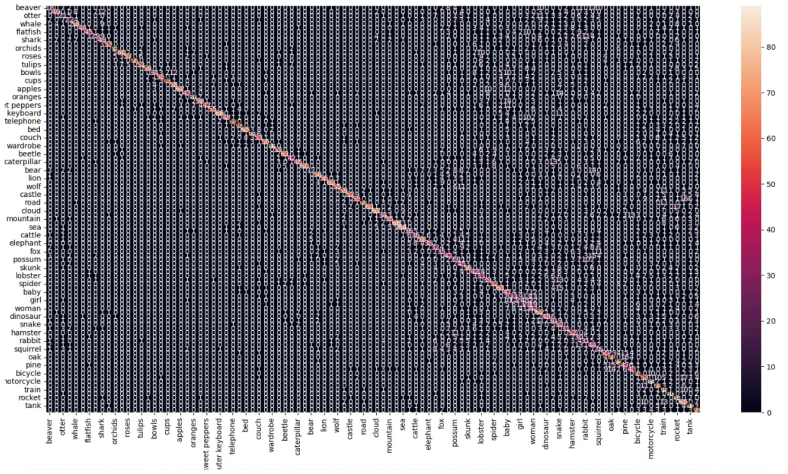


Figure 7.2: Confusion matrix for 5 sequential tasks for superclass order.

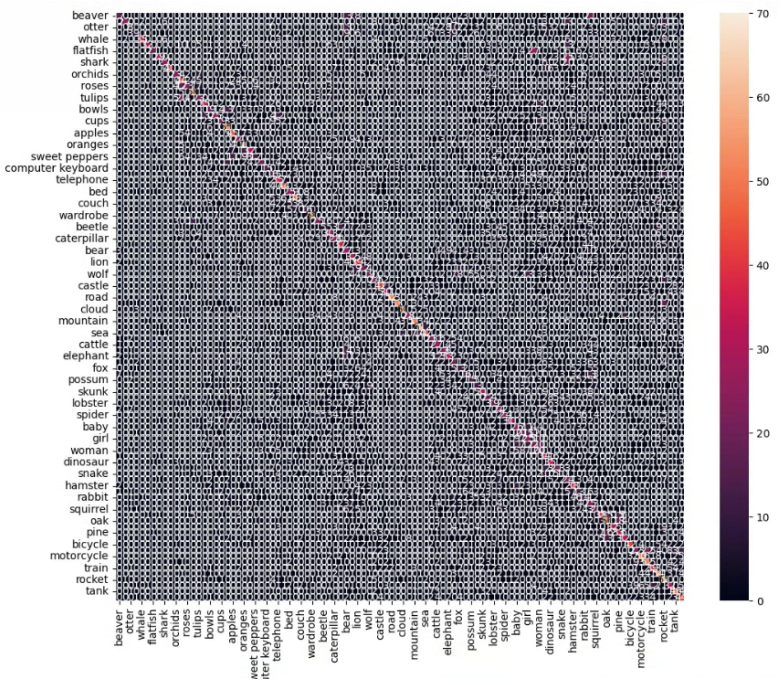


Figure 7.3: Confusion matrix for 9 sequential tasks for superclass order.

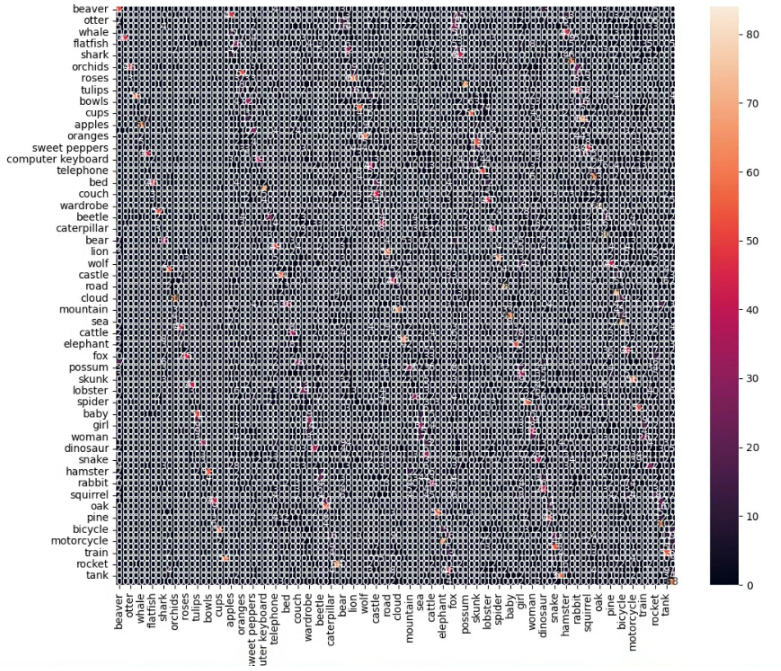


Figure 7.4: Confusion matrix for 19 sequential tasks for superclass order.

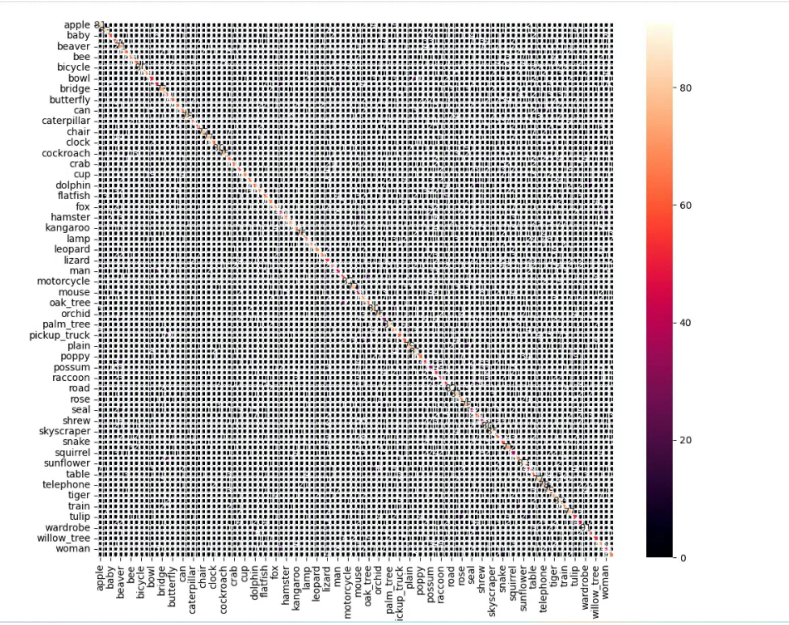


Figure 7.5: Confusion matrix for 1 sequential tasks for random order.

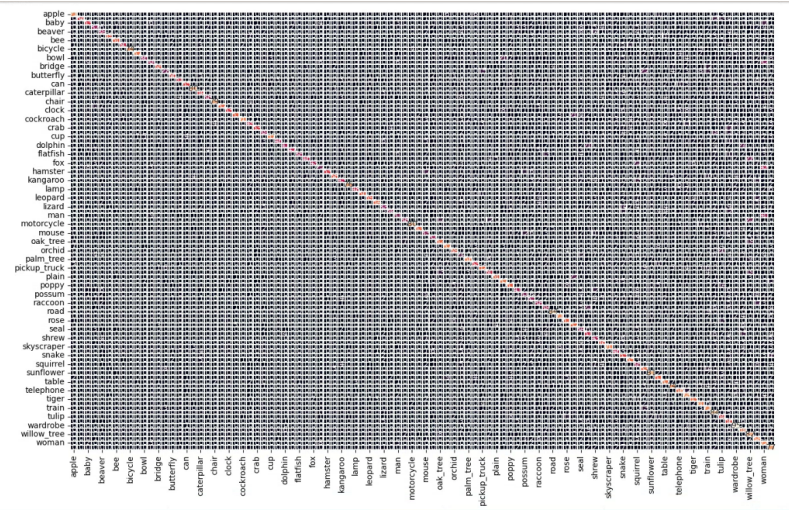


Figure 7.6: Confusion matrix for 5 sequential tasks for random order.

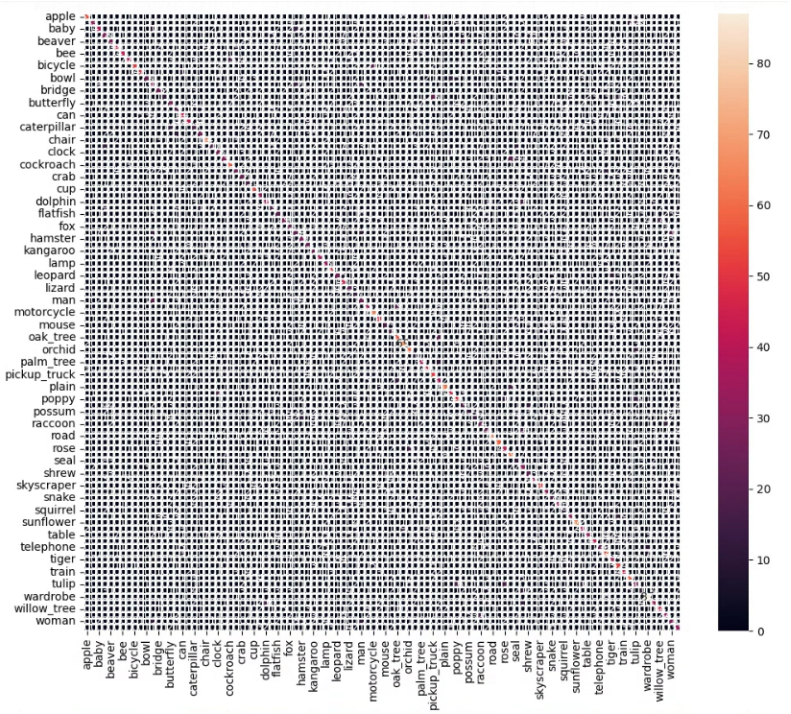


Figure 7.7: Confusion matrix for 9 sequential tasks for random order.

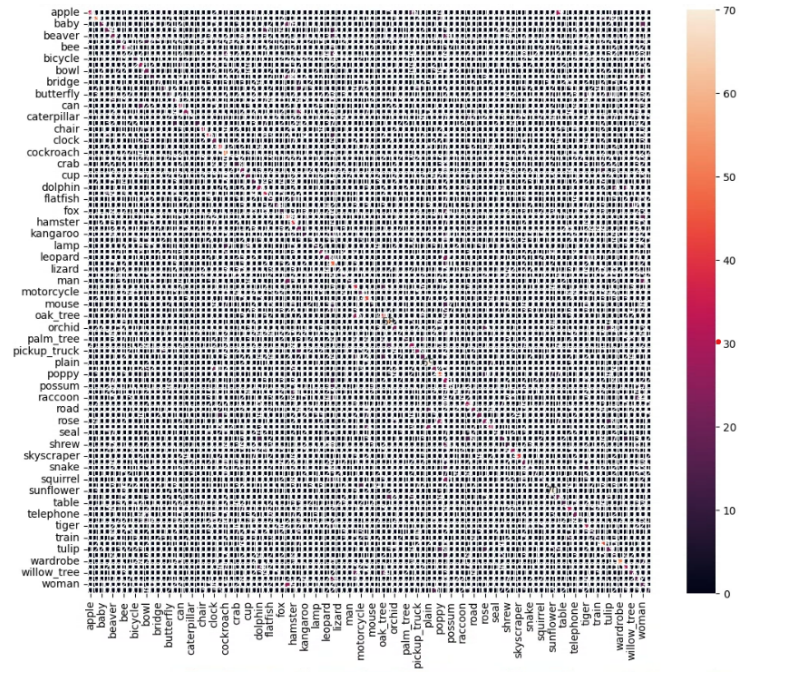


Figure 7.8: Confusion matrix for 19 sequential tasks for random order.

Classification accuracy for each tasks:

Figure 7.9, Figure 7.10 show the classification accuracy results at each incremental step for new classes only.

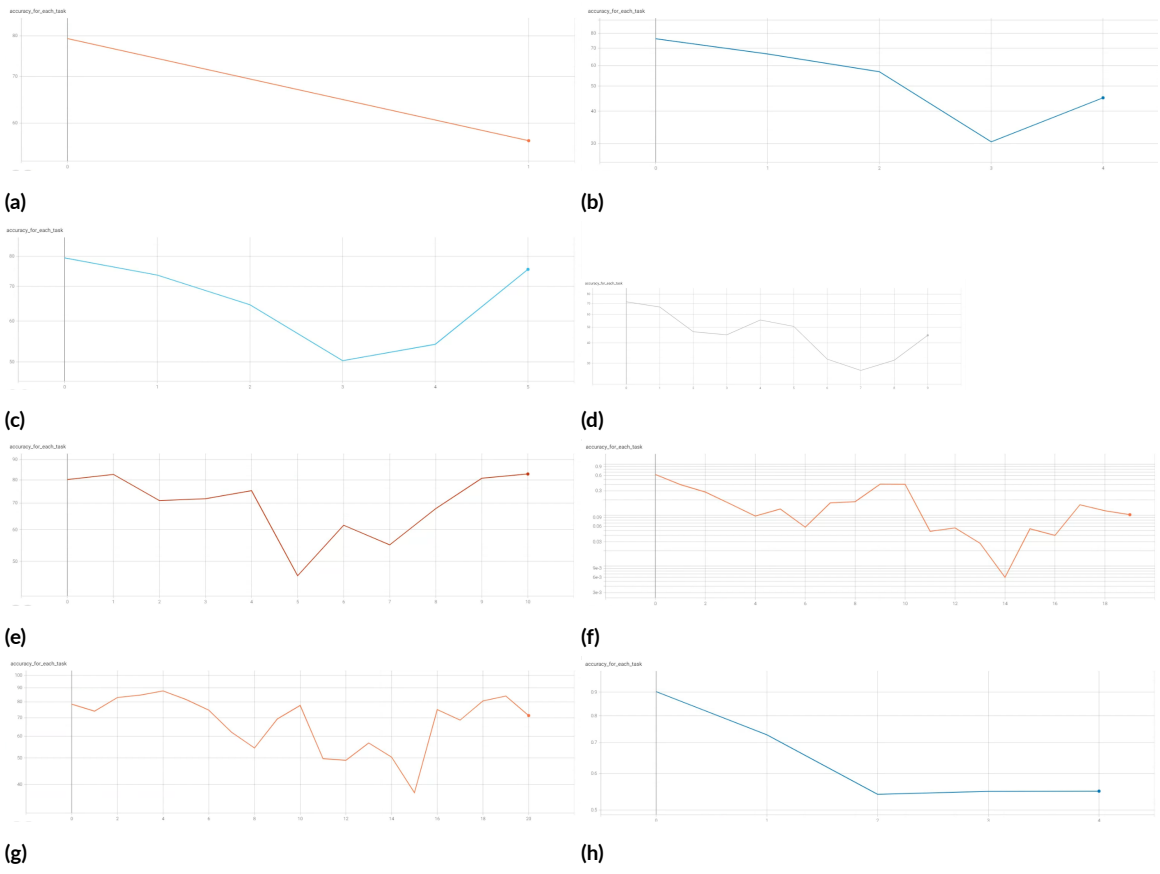


Figure 7.9: Classification accuracy at each step for 1, 4, 5, 9,10, 19, 20 sequential tasks for superclass order and 4 one class per supeclass

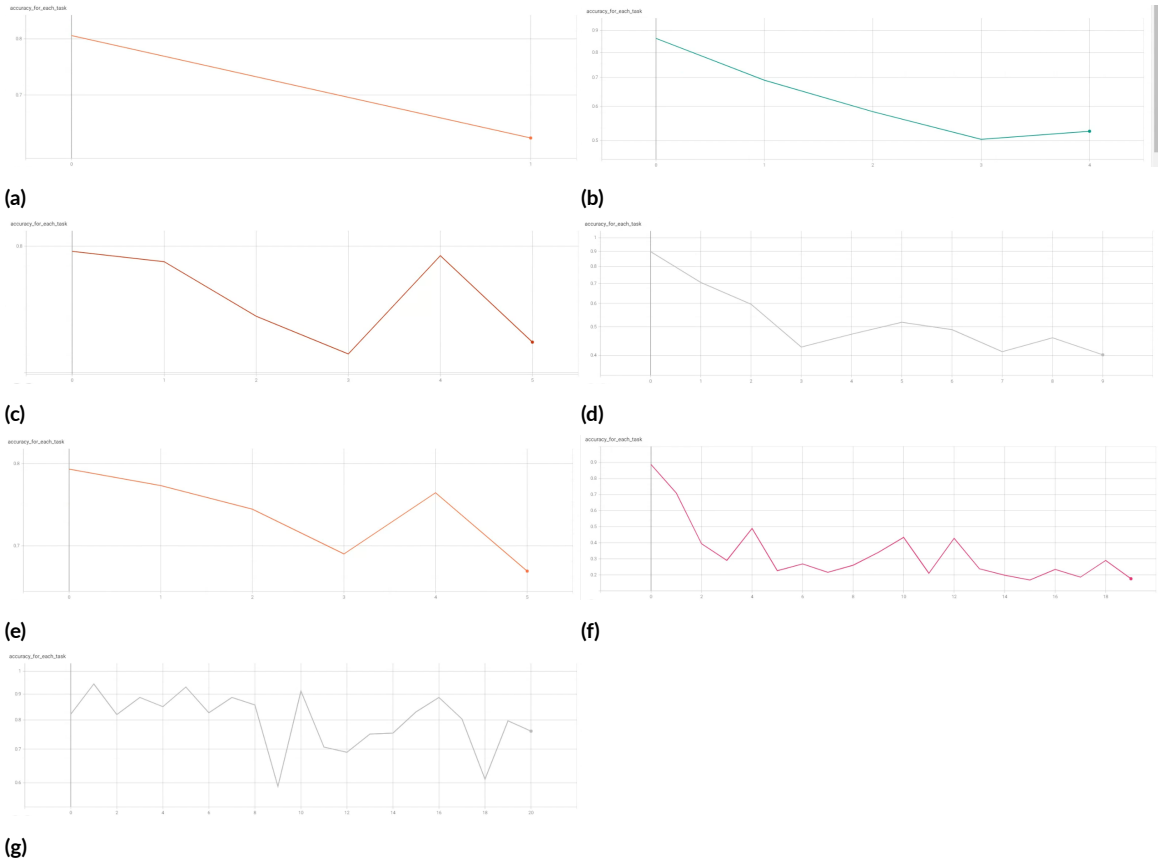


Figure 7.10: Classification accuracy at each step for 1, 4, 5, 9,10, 19, 20 sequential tasks for random order

Forgetting result:

Figure 7.11, Figure 7.12 show the forgetting results at each incremental step for previously learned classes.

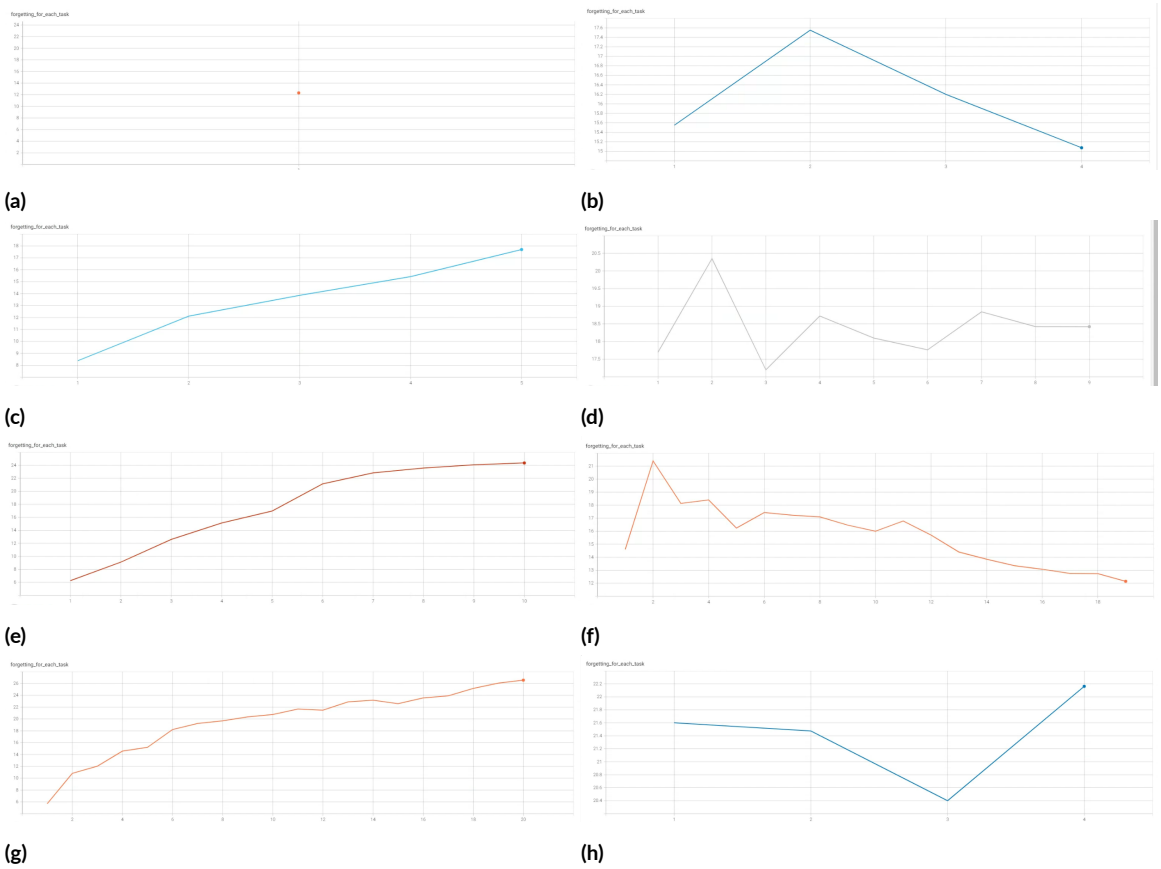


Figure 7.11: Forgetting result at each step for $1, 4, 5, 9, 10, 19, 20$ sequential tasks for superclass order and 4 one class per supeclass

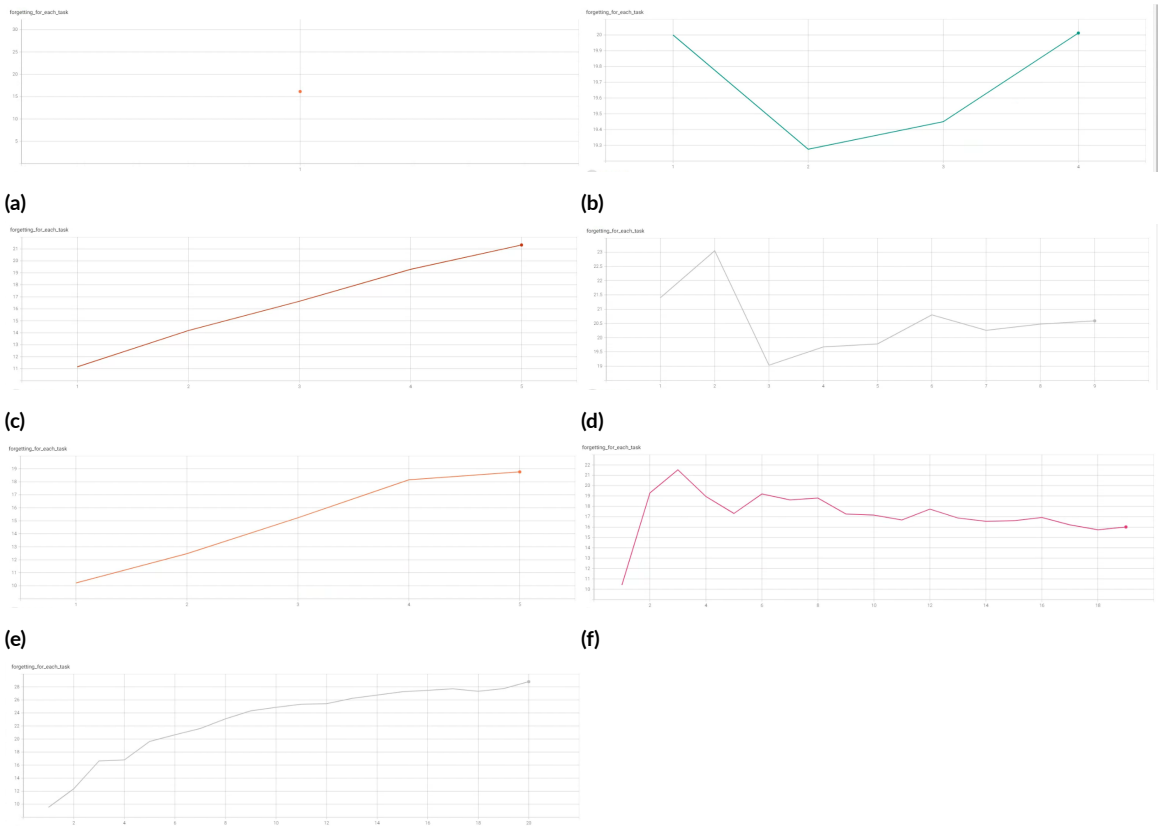
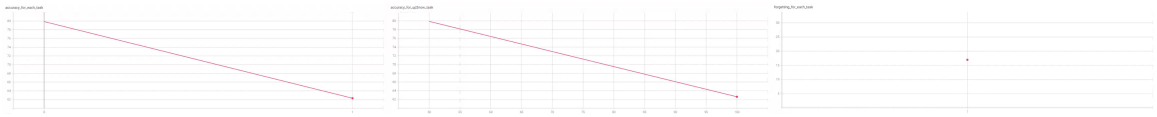


Figure 7.12: Forgetting result at each step for 1, 4, 5, 9,10, 19, 20 sequential tasks for random order

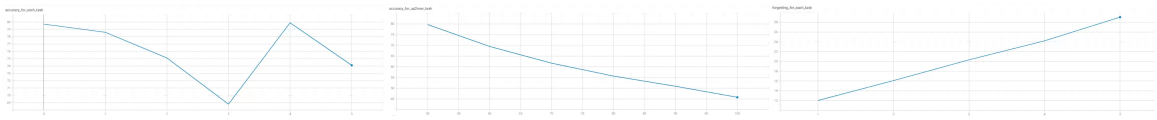
Adding Cluster loss term results:

The figures below visualize classification accuracy at each step on the previously learned classes, classification accuracy of each step and forgetting result at each task on old classes. we provide the results for tasks 1,5,10,19, 20 after adding cluster loss term to the loss function.



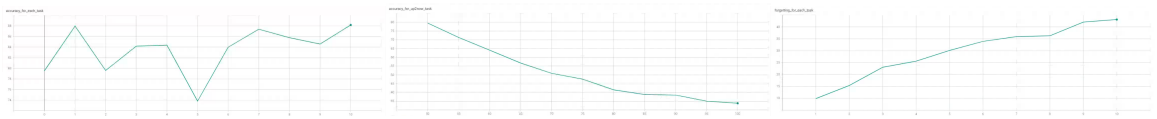
(a) (b) (c)

Figure 7.13: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 1 with random class ordering



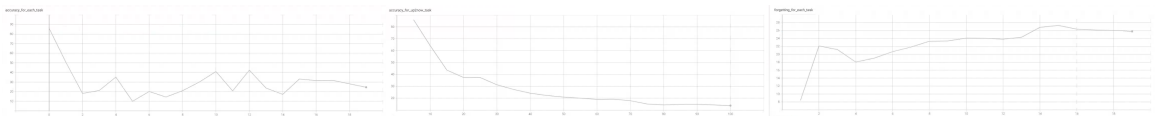
(a) (b) (c)

Figure 7.14: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 5 with random class ordering



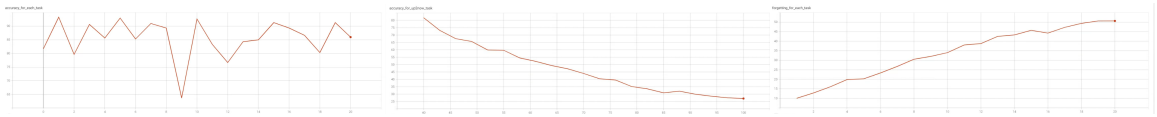
(a) (b) (c)

Figure 7.15: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 10 with random class ordering



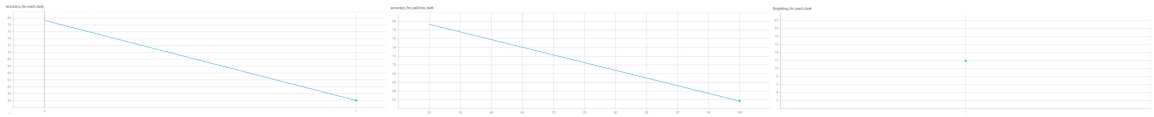
(a) (b) (c)

Figure 7.16: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 19 with random class ordering



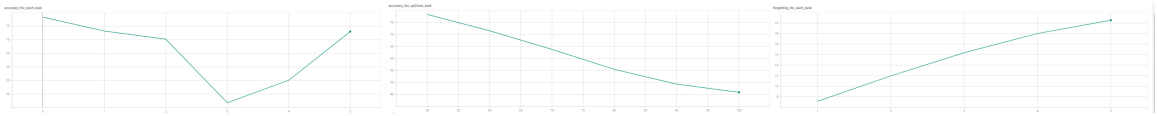
(a) (b) (c)

Figure 7.17: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 20 with random class ordering



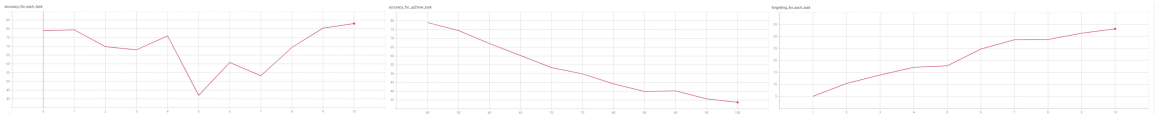
(a) (b) (c)

Figure 7.18: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 1 with superclass ordering



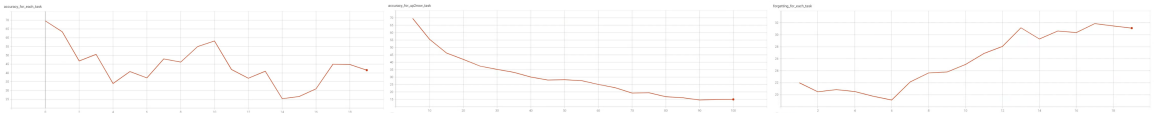
(a) (b) (c)

Figure 7.19: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 5 with superclass ordering



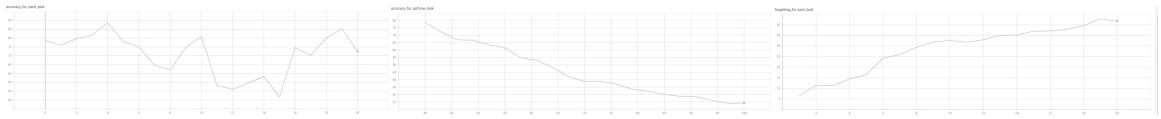
(a) (b) (c)

Figure 7.20: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 10 with superclass ordering



(a) (b) (c)

Figure 7.21: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 19 with superclass ordering



(a) **(b)** **(c)**

Figure 7.22: Classification accuracy at each step, classification accuracy and forgetting result on old classes for task 20 with superclass ordering