Incremental Learning in Image Classification

Tommaso Monopoli
Politecnico di Torino
s278727

Ruggiero Francavilla
Politecnico di Torino
s278312

Abstract

Incremental learning is a machine learning paradigm in which a model is trained continuously on new input data whenever it is available, extending the existing model's knowledge. In our dissertation, we experiment on three incremental learning methods, Finetuning, Learning without Forgetting and iCaRL. Our study then focuses more in-depth on the iCaRL framework, which makes use of the techniques of knowledge distillation and prototype rehearsal to preserve past knowledge acquired by a deep convolutional neural network, while simultaneously training on new classes. We then perform an ablation study, experimenting with different combinations of distillation and classification losses and introducing new classifiers. By inspecting the behavior of the newly introduced elements, we seek to better understand how the model benefits from each of its individual components and what its possible weaknesses are. We subsequently propose some variations of the original iCaRL algorithm that attempt to tackle such issues, and we verify their effectiveness. For a fair comparison, we perform our tests on CIFAR-100, as used in the original iCaRL paper. Eventually, our extensions of iCaRL lead to a model achieving a better accuracy than iCaRL on CIFAR-100.

1. Introduction

Incremental learning Human learning is naturally incremental: what is commonly referred to as "learning" is a continuous act of gathering experiences, processing incoming information and enriching knowledge accumulated in the past with new one. However, in the machine learning setting, it is often assumed that all the data a model will ever need to learn from is available from the very start of the algorithm's lifetime. While this assumption can sometimes be reasonable, some applications may require machine learning models to grow over time. More specifically, in the field of computer vision, we wish for a model to recognize new categories of images without forgetting the previously learned ones; in other words, we would like to design a machine learning algorithm that is capable of being trained on a continuous stream of data and "incrementally" learn to classify new classes of images while also maintaining the memory of classes seen in the past. We call this scenario *class-incremental learning*.

In [13], Rebuffi *et al*. proposed a list of 3 properties an algorithm should fulfill in order to qualify as class-incremental:

1. it should be trainable from a stream of data in which examples of different classes occur at different times,
2. it should at any time provide a competitive multi-class classifier for the classes observed so far,
3. its computational requirements and memory footprint should remain bounded, or at least grow very slowly, with respect to the number of classes seen so far.

While the first two points constitute the definition of incremental learning itself, the third one rules out trivial solutions to the problem, such as continuously storing all training samples and re-training a model from scratch as soon as new classes of data are available. This last approach is commonly referred to in the literature as "joint training".

Joint training constitutes an upper bound on the performances of every possible incremental learning method. However, there are instances in which the joint approach is simply not feasible: for example, when past data is no longer available, or when a complete re-train of a classifier would be overly expensive in terms of computation workload or memory footprint. This motivates the search for incremental algorithms which are able to infer knowledge from the new data available and retain past knowledge without having to rely on the entirety of the old training samples.

Catastrophic forgetting All algorithms that attempt to accomplish the task of incremental learning must face the problematic phenomenon commonly referred to as *catastrophic forgetting*. With the term catastrophic forgetting we mean the scenario in which, once an incremental model learns new classes, it experiences a performance drop in classifying old ones: in other words, it starts forgetting past knowledge.

Our setting In the last few years, several methods have been proposed to counteract the catastrophic forgetting phenomenon: in the next sections, we present three of them that are well known in the literature and address this issue in different ways. All of said algorithms respect the three aforementioned principles of incremental learning and succeed in the task to various degrees. We focus our dissertation on the iCaRL learning framework [13], currently considered among the state-of-the-art algorithms in incremental learning for image classification and used as baseline for several recent proposals in the field [5]. The dataset we employ for our experiments is CIFAR-100 (the same used in [10]), which is composed of $60,000$ 32×32 pixel images belonging to 100 classes (we further describe it in section 2.1). We simulate an incremental training scenario by dividing the dataset in batches of 10 classes each and training our models on the resulting 10 batches in an incremental fashion. We call the training on a single batch of 10 classes an "incremental training step". After each incremental training step, we test the model's performance on all the classes it has been trained on so far; that is, after the k-th incremental training step, we test on $10 \times k$ classes, and compute various statistics which we subsequently report and discuss. For consistency across different experiments and to allow a fair comparison between the learning methods described, the way the 100 classes of CIFAR-100 are distributed in the 10 batches of 10 classes is fixed by setting the same random seed in all the runs. Our source code is available at `https://github.com/RuggieroFrancavilla/Incremental-Learning-iCaRL/`.

2. Baselines

In this section we describe our implementation of the three baseline models, and we discuss the results achieved by each of them on CIFAR-100.

2.1. Dataset, model architecture, data preparation

All the experiments described in this paper are based on the CIFAR-100 dataset [9]. CIFAR-100 consists of 60000 32×32 RGB images in 100 non-overlapping classes, with 600 images per class (500 for training and 100 for testing). There are 50000 training images and 10000 test images in total.

We randomly divide the dataset in 10 batches of 10 classes. This is done with a fixed seed, so that the random shuffling of the 100 classes is the same across all the experiments performed. These batches are presented to the learning algorithm one at a time: during each training phase, the model learns from the 5000 images of the batch. After each of the 10 incremental training steps, the resulting model is evaluated on a test dataset composed of all the test images of the classes seen so far (a total of $1000 \times i$ images, where i is the incremental training step).

Our aim is to reproduce the results achieved on CIFAR-100 by three class-incremental methods presented in paper [13]: *finetuning*, *Learning without Forgetting* and *iCaRL*. We rely on the *pytorch*[1] framework to implement these baselines.

All our experiments involve the incremental training of a 32-layers ResNet [3] with 100 output neurons, the same used by the original iCaRL implementation.

2.2. Finetuning

The first baseline approach we consider is the Finetuning, briefly explained in [11]. Finetuning learns an ordinary multi-class network without taking any measures to prevent catastrophic forgetting. It can also be interpreted as learning a multi-class classifier for new incoming classes by finetuning the previously learned multi-class classification network. The algorithm makes use of a convolutional neural network, intended as a trainable feature extractor φ. The CNN is followed by a single fully connected classification layer with 100 sigmoid output nodes, as the final number of classes to learn. All features vectors and results of any operation on feature vectors are L^2-normalized. The parameters Θ of the network are split into the parameters for the feature extraction part and the weight vectors $w_1, ..., w_{100}$ of the last fully connected classification layer. The resulting network outputs are, for any class $y \in \{1, ..., 100\}$,

$$g_y(x) = \frac{1}{1 + exp(-a_y(x))} \text{ with } a_y(x) = w_y^T \varphi(x) \quad (1)$$

To predict a label y^* for an image x, the usual classification rule for a neural network can be used:

$$y^* = \arg\min_{y=1,...,t} g_y(x)$$

where t is the number of classes seen so far by the algorithm.

2.3. Learning Without Forgetting

As a first approach to tackle the catastrophic forgetting phenomenon, Rebuffi *et al*. [13] propose a learning strategy based on *knowledge distillation*, borrowing ideas from Hinton *et al*. [4] and Li and Hoiem [11].

Distillation knowledge consists in transferring knowledge from a model to another one. Alongside the usual classification loss, a regularization term called *distillation loss* is added to it, in order to prevent catastrophic forgetting. While the classification loss encourages the network to perform correct prediction on new classes, the distillation loss tries to preserve the the outputs of the past configuration of the network. In particular, the distillation loss aims at

[1] `https://pytorch.org/`

preserving the responses given by the instance of the model learned in the previous incremental training step (i.e. before updating the parameters) on the output neurons associated to previously seen classes. Therefore, the parameters Θ are updated according to the output produced both by the past configuration of the network and the actual one being further trained.

For each image x_i in the training set with label y_i, the per-sample loss function computed is:

$$l_i(\Theta) = -\Big[\sum_{y=s}^{t} \delta_{y=y_i} \log g_y(x_i) + \delta_{y\neq y_i} \log(1 - g_y(x_i)) + \\ + \sum_{y=1}^{s-1} q_i^y \log g_y(x_i) + (1 - q_i^y) \log(1 - g_y(x_i))\Big]$$

where classes from s to t are the currently seen ones and classes from 1 to $s-1$ are the old classes; $g_y(x_i)$ is the network output of the image x_i for the class y; q_i^y is the output of the old copy of the model given the image x_i for the class y; $\delta_{y=y_i}$ is the Kronecker delta.

The total loss is computed as the mean of the per-sample losses over all the training images seen during an incremental training step, at the end of each epoch:

$$L(\Theta) = \frac{1}{|D|} \sum_{(x_i,y_i)\in D} l_i(\Theta) \tag{2}$$

where D is the dataset composed of all the images of the currently seen 10 classes, on which the network is trained at that incremental step.

2.4. iCaRL

Building on top of the previously described baseline, Rebuffi *et al*. [13] introduce other two main components:

- *representation learning* using both knowledge distillation (previously described) and *herding-based prototype rehearsal*,
- classification by a *nearest-mean-of-exemplars* (NME) rule

For each learned class, the model chooses some images of that class to store in memory. We call these images *exemplars*. At each training phase, the model is trained on the concatenation of the images of the new classes which are currently being seen and the exemplars of the classes seen previously. Moreover, exemplars are also exploited to approximate the class means in feature space (used by NME at classification time); the feature space is defined by the mapping function $\varphi(\cdot)$, which includes all the layers of the ResNet32 architecture apart from the last fully connected layer; As the memory usage of the algorithm must remain bounded, at most K exemplars can be stored at any given time. If the current number of classes learned by the model is N, then the number of stored exemplars for each class is $m = K/N$. We refer to the list of the stored exemplars for the i-th seen class as "exemplar set" for that class, and we denote it by P^i. Moreover, we denote the list of all the exemplar sets by $\mathcal{P} = (P^1, \ldots, P^{s-1})$.

We will now describe more specifically the features introduced by iCaRL, in the order they are encountered during the incremental training and evaluation process of iCaRL.

Representation Learning The first step of the training method for iCaRL is the representation learning step. Whenever iCaRL obtains data, $X^s, \ldots, X^t$ for new classes, $s, \ldots, t$, it updates the learnable parameters of the feature extractor φ and constructs exemplar sets $P^s, \ldots, P^t$. First, iCaRL constructs an augmented training set consisting of the currently available training examples together with the stored exemplars:

$$\mathcal{D} \leftarrow \bigcup_{y=s,\ldots,t} \{(x,y) \mid x \in X^y\} \quad \cup \quad \bigcup_{y=s,\ldots,t} \{(x,y) \mid x \in P^y\}$$

Note that in the first incremental training step there is no such concept as "old classes", since none have been seen before, therefore the training set $\mathcal{D}$ is composed only of the union of $X^1, \ldots, X^{10}$. Next, the current network is evaluated for each example and the resulting network outputs for all previous classes ($y = 1, \ldots, s-1$) are stored:

$$q_i^y \leftarrow g_y(x_i) \text{ for all } (x_i, \cdot) \in \mathcal{D}$$

Finally, the network parameters are updated by minimizing the loss function introduced in the Learning without Forgetting section 2: minimizing the classification loss encourages the network to output the correct class for new classes, whereas minimizing the distillation loss encourages the network to reproduce the scores q_i^y of the network computed before the current representation learning step. The augmentation of the training set with the stored exemplars, as well as the distillation loss, are two modifications that aim at preventing or at least mitigating catastrophic forgetting, by ensuring that at least some information about the data distribution of all the previous classes enters the training process. Standard end-to-end learning methods, such as Stochastic Gradient Descent (SGD) with backpropagation, can be used to perform this training step. We use the expression "prototype rehearsal" to indicate that the model learns from exemplar images of the previously seen classes multiple times throughout the incremental training process.

Exemplar management Once the model's feature representation is updated on new training classes, two routines

are called to manage the exemplar sets: one selects the exemplars for new classes (with the so-called *herding* procedure) and one reduces the size of the exemplar sets of previous classes to fit within the K upper bound.

The first routine to be applied is the reduction of exemplar sets of previously seen classes. It is particularly simple: to reduce the number of exemplars from any m' to m, one discards the exemplars $p_{m+1}, \dots, p_{m'}$, keeping only the exemplars $p_1, \dots, p_m$.

The second routine is the exemplar selection step. As no more than K exemplars must be stored at any time, this procedure selects $m = K/t$ exemplars (up to rounding) for each new class, where t is the number of classes observed so far. Given the whole image set $X^y = (x_1, \dots, x_n)$ of class y seen during the representation learning step, the feature vectors $\varphi(x_i)$ are computed for each image, then the average feature vector over all examples of X^y is computed:

$$\mu_y = \frac{1}{n} \sum_{x \in X^y} \varphi(x) \tag{3}$$

Note that all feature vectors are L^2-normalized, and the results of any operation on feature vectors, e.g. averages, are also re-normalized, which we do not write explicitly to avoid a cluttered notation.

We refer to the average feature vector μ_y as *class mean* of class y. Exemplars $p_1, \dots, p_m$ are selected and stored iteratively. The selection method chosen is the so-called *herding* technique: in each k-th step of the iteration, one more example p_k of the current training set is added to the exemplar set P^y, namely the one that causes the average feature vector over all images of X^y already in the exemplar set to best approximate the actual class y mean:

$$p_k \leftarrow \operatorname*{arg\,min}_{x \in X^y \setminus P^y} \left\| \left(\mu_y - \frac{1}{k} \left[\phi(x) + \sum_{j=1}^{k-1} \phi(p_j) \right] \right) \right\|$$

Thus, herding causes each exemplar "set" to be really a prioritized list: exemplars earlier in the list are more important. For this reason, the exemplar set reduction routine can simply perform a "cut" in the list, keeping only the m "most relevant" exemplars for each class. Notice that there won't be any duplicate images in P^y at the end of this routine, as candidate exemplars are selected among those images in X^y which are not already in P^y.

Nearest-mean-of-exemplars classifier iCaRL uses a *nearest-mean-of-exemplars* (NME) classification strategy, using the CNN model just as a feature extractor. To predict a label y^* for an image x, it computes an average feature vector for each class observed so far, $\overline{\mu}_1, \dots, \overline{\mu}_t$ where

$$\overline{\mu}_y = \frac{1}{|P^y|} \sum_{p \in P_y} \varphi(p) \tag{4}$$

is the average feature vector of all exemplars for a class y, and the feature vector $\varphi(x)$ of the image that should be classified. Then, the predicted class is the class y^* which minimizes the L^2 distance between $\varphi(x)$ and $\overline{\mu}_y$:

$$y^* = \operatorname*{arg\,min}_{y=1,\dots,t} \left\| \varphi(x) - \overline{\mu}_y \right\|$$

As explained in iCaRL's original paper, this classification approach tackles the problem of the weight vectors $w_1, \dots, w_t$ of the output fully connected layer being decoupled from the feature extraction routine φ. In contrast, NME classifier doesn't use weight vectors to perform a classification; instead, the average feature vectors $\overline{\mu}_1, \dots, \overline{\mu}_t$ change consistently with φ, making the NME classifier robust against changes of the feature representation, which happen during the representation learning step 2.4.

Basically, the NME classifier can be seen as a 1-nearest neighbor classifier (KNN with K=1) in the feature space defined by φ. The data points used by this KNN to perform classification are the average feature vectors $\overline{\mu}_1, \dots, \overline{\mu}_t$. $\varphi(x)$ is assigned the label y^* of the nearest neighbor μ_{y^*} (euclidean distance is used).

Notice that, since $\overline{\mu}_1, \dots, \overline{\mu}_t$ provide at any time good approximations of the actual class means of the seen classes (thanks to the herding-based exemplars selection 2.4), the NME classifier should perform in a similar way to a nearest class mean (NCM) classifier, in which the actual class means are used in place of the average feature vectors computed from the t available exemplar sets. By the way, the NCM classifier is not adherent to the incremental learning requirement of keeping the memory footprint of the model bounded, since it requires all the training images seen so far to be stored in order to compute the actual class means. Therefore, NME represents a feasible alternative to NCM.

A useful "trick" After an incremental training step, in which new classes from s to t have been seen, iCaRL has constructed new exemplar sets $P^s, \dots, P^t$ and can then estimate the t class means needed for the NME classifier from the stored exemplars. However, we notice that, at the very little cost of keeping $t - s$ feature vectors in memory, we can keep the actual class means computed with (3) in the exemplar selection step and use them at classification time in place of their approximations given by (4). In this way, we expect the classification accuracy on classes $s, \dots, t$ to be slightly better than with class mean approximations used in the "standard" NME. Our implementation of iCaRL and all the proposed modifications of it which employ the NME as classifier make use of this "trick".

2.5. Results

In this section we introduce a testing protocol which we used for evaluating incremental learning methods and models.

Given the CIFAR-100 dataset, the 100 classes are arranged in a fixed random order and divided into 10 batches of 10 classes each (as explained in 2.1). The method is then trained in a class-incremental way on the available training data. After each batch of classes, the resulting model is evaluated on the portion of test images of CIFAR-100 associated to the already seen classes. Note that, even though test data is used more than once, overfitting cannot occur since testing results are not revealed to the learning method. The result of the evaluation are curves of the classification accuracies after each batch of classes, along with the average of these accuracies, called *average incremental accuracy*.

We evaluate the methods of *Finetuning* (2.2), *Learning without Forgetting* (2.3) and *iCaRL* (2.4) using this testing protocol. In particular, we will evaluate three classifiers for iCaRL: the classification layer (1) built on top of the feature extractor CNN (iCaRL.CNN), the Nearest-mean-of-exemplars classifier (iCaRL.NME) and the Nearest-class-mean classifier (iCaRL.NCM).

The hyperparameters used in training are the ones provided in [13], and are reported hereafter. For all the methods to be tested, we train a 32-layers ResNet with 100 output neurons, allowing iCaRL to store up to $K = 2000$ exemplars. Each training step consists of 70 epochs. The learning rate starts at 2.0 and is divided by 5 after 49 and 63 epochs (7/10 and 9/10 of all epochs). We train the network using standard SGD with minibatches of size 128 and L^2 weight regularization with weight decay parameter of 10^{-5}.

Some image preprocessing steps are applied to training and test images. Training images are padded at the borders, then a central 32×32 crop is taken, then horizontally flipped with probability 1/2 and finally standardized, subtracting the per-channel mean and dividing by the per-channel standard deviation of the training split of CIFAR-100. The same preprocessing is performed on stored exemplars. As for test images, we just subtract the said channel-wise mean and standard deviation from them.

The evolution of test accuracies for each method are reported in tab. 1 and plotted in fig. 1.As expected, the finetuning approach achieves the worst results, as it takes no actions to prevent catastrophic forgetting. The behavior of this algorithm is displayed in the confusion matrix in fig. 2. Finetuning is essentially unable to remember past classes and labels input images with classes only from the last batch of classes seen.

The notable improvement achieved by the Learning without Forgetting method (+17.8% on the last batch of classes) proves the crucial role of the knowledge distillation process in containing catastrophic forgetting. This is evident in the confusion matrix in fig. 3. The choice of the classification and distillation losses is definitely relevant, but not unique: we shall discuss and try out different losses

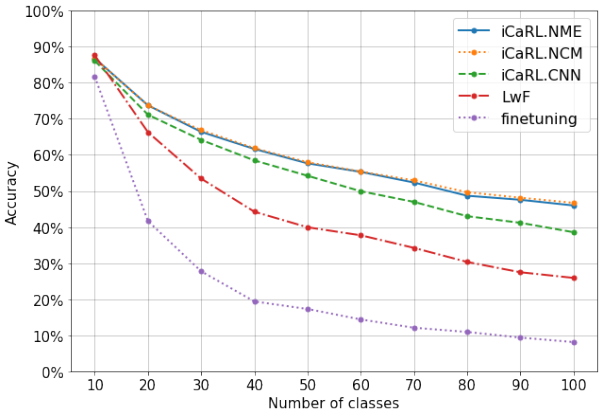

Figure 1. Evolution of the test accuracies over the number of classes incrementally seen by the baseline methods

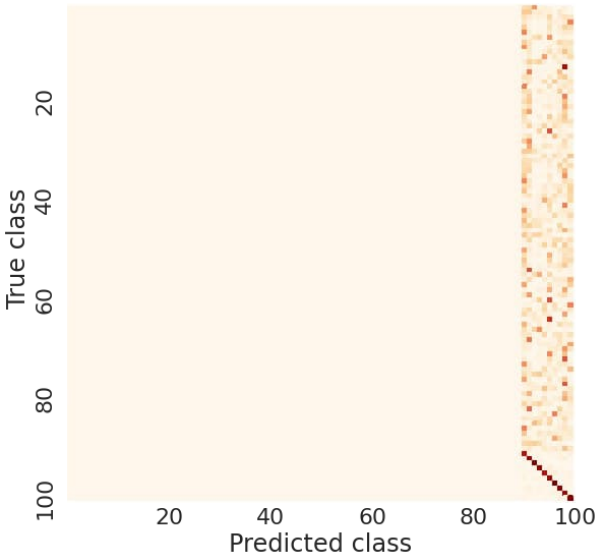

Figure 2. Confusion matrix of the Finetuning baseline. Catastrophic forgetting is clearly visible.

in section 3, and compare performances to those achieved with binary cross-entropy loss used by iCaRL.

iCaRL.CNN has essentially the same configuration as LwF, with the exception of using exemplars for reviewing past knowledge. The introduction of this new element in the framework further mitigates catastrophic forgetting and results in a +12.7% test accuracy on the last batch of classes. The confusion matrix for this baseline is displayed in fig. 4.

Finally, the introduction of the NME classifier completes the iCaRL setup and provides a noticeable performance improvement (+7.4% on the last batch of classes, with respect to iCaRL.CNN, and only -0.7% with respect to iCaRL.NCM, which we can consider as an upper bound of iCaRL.NME). The confusion matrix for iCaRL.NME is displayed in fig. 5, and the one of iCaRL.NCM in fig. 6.

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NCM	0.868	0.737	0.669	0.619	0.580	0.554	0.529	0.497	0.481	**0.467**	**0.600**
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	0.460	0.596
iCaRL.CNN	0.861	0.712	0.642	0.585	0.542	0.499	0.470	0.430	0.412	0.386	0.554
LwF	0.876	0.663	0.534	0.443	0.399	0.377	0.342	0.303	0.275	0.259	0.447
Finetuning	0.817	0.417	0.277	0.194	0.173	0.144	0.121	0.109	0.094	0.081	0.243

Table 1. Test accuracies achieved by the baseline models at each incremental step (over the number of classes

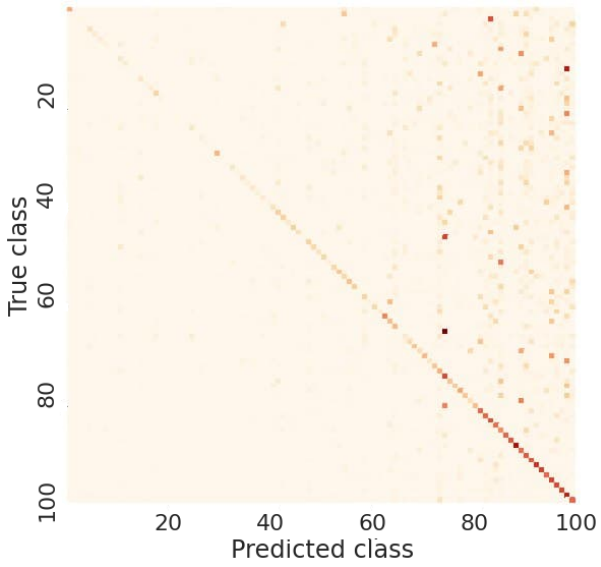

Figure 3. Confusion matrix of the LwF baseline.

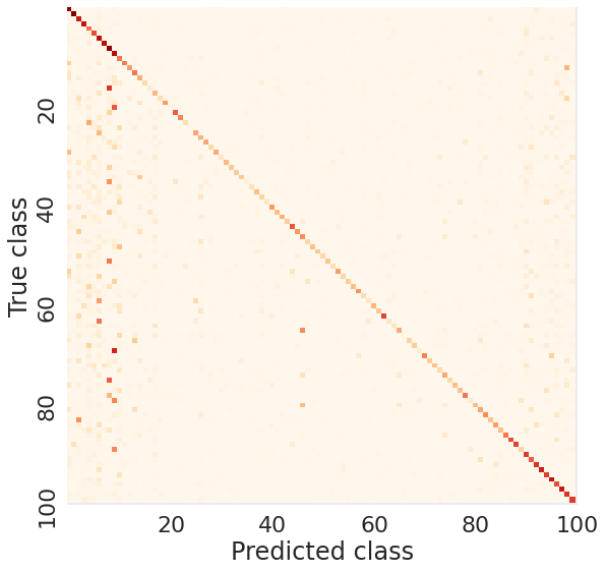

Figure 4. Confusion matrix of iCaRL.CNN.

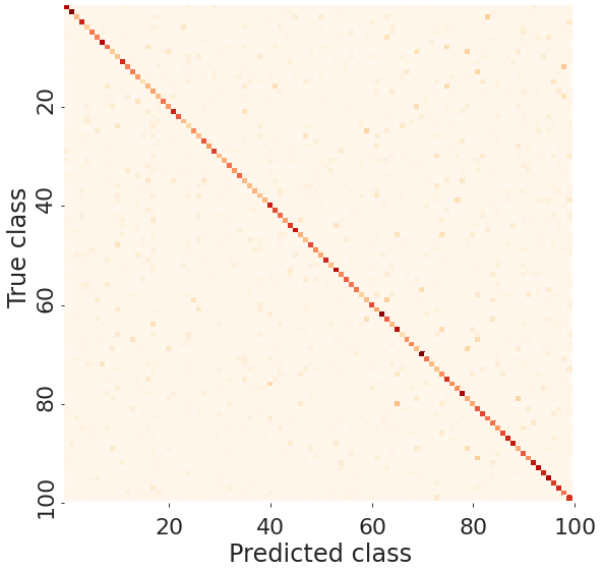

Figure 5. Confusion matrix of iCaRL.NME.

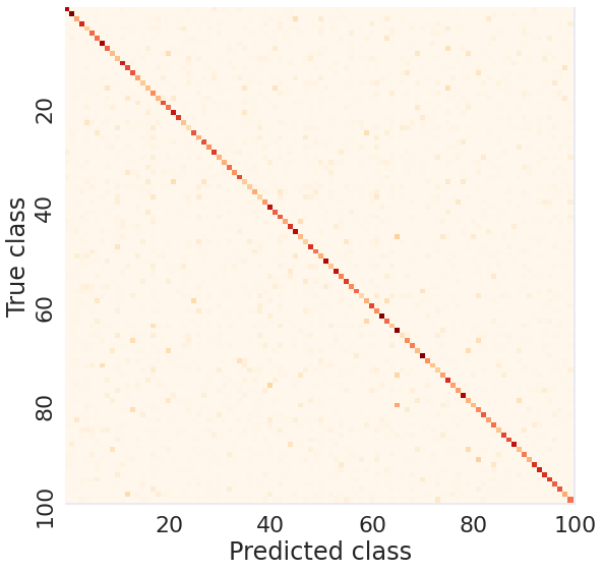

Figure 6. Confusion matrix of iCaRL.NCM.

This remarkable result shows that the use of different classifiers on top of the architecture can sensibly influence the model's performance: we shall more widely discuss this topic in section 4, and propose some variations on the classifiers used.

3. Ablation study on losses

The choice of the loss function both for classification and distillation is not unique: while the original iCaRL framework uses the binary cross-entropy loss both as classification and distillation loss, variations are possible. To evaluate the impact of the particular choice of the classification and distillation losses on iCaRL's performances, we train iCaRL again experimenting with different combinations of losses.

3.1. Different losses combinations

3.1.1 Theoretical description

As a first experiment, different combinations of classification and distillation losses are executed.
The tested losses for the classification component are:

1. Cross Entropy Loss;
2. MSE Loss with Softmax.

With all these different losses, we achieved good performances (fig. 2), but worse in comparison with iCaRL ones. Therefore, we decided not to make use of a different loss for the classification term, and keep the Binary Cross Entropy loss, to have a direct comparison when we test different losses as distillation term, and analyze the effects obtained on the task.

The tested losses for the distillation component are:

- L1 Loss with Softmax:

$$l_d(\Theta) = \sum_{y=1}^{s-1} |\hat{q}_i^y - \hat{g}_y(x_i)|$$

- MSE Loss with Softmax:

$$l_d(\Theta) = \sum_{y=1}^{s-1} (\hat{q}_i^y - \hat{g}_y(x_i))^2$$

where $\hat{q}_i^y$ and $\hat{g}_y(x_i)$ are respectively the softmax outputs of the old copy of the network and of the current network. The softmax function is used to normalize the output of the network to a probability distribution over predicted output classes. When applying a softmax function to a vector element-wise, we obtain a probability vector whose probabilities mirror the proportions of the values found on the input vector.

3.1.2 Experiments and Results

We train iCaRL two more times, employing the L_1 and MSE loss as distillation terms respectively. In tab. 3 we report the evolution of the test accuracies achieved by the iCaRL baseline and its variations with respect to distillation loss. As we can notice from the table, when using the L_1 loss as distillation loss the achieved test accuracies are lower than iCaRL's results at early stages, but later ones seems closer to the ones achieved by iCaRL. On the other hand, the MSE loss as distillation term exhibits the nearly the same results as iCaRL in early incremental training steps, but at the end results are slightly lower.

The reason of these behaviors can be found in the definitions of L_1 loss and MSE loss. Since we are working with normalized softmax outputs, the found distances by these two methods will likely assume a small value $<= 1$. MSE loss formula, using squared values also for the expected similarity of outputs of the new and old network, makes obtained values even lower, since we are talking of values smaller than one. Therefore, $L1$ loss gives an higher contribute, in terms of distillation phenomena to the overall loss, in comparison to MSE loss. This implies that the implementation of iCaRL with MSE loss achieves good results on new classes, but it is much worse than the implementation with $L1$ in correctly classifying old classes, and provides and explanation as to why implementation with MSE distillation loss is more performing with fewer classes. On the other hand, $L1$ loss being more coherent with the final results, shows distillation more effective process results.

The uniformity of behaviour of L_1 loss, with respect to the behaviour of MSE loss is clearly observable also from the resulting confusion matrices (fig. 7 and 8 respectively).

3.2. Temperature Softmax Distillation

3.2.1 Theoretical Description

We attempt a different form of distillation loss as proposed by Hinton *et al*. [4] in the context of knowledge distillation from large to small models. The first to use this loss in incremental learning framework were Li and Hoiem [11]. Given a generic training image x_i, the loss is defined as:

$$l_d(\Theta) = -\sum_{y=1}^{s-1} \hat{q}_i'^y \log \hat{g}'_y(x_i)$$

where $\hat{q}_i'^y$ and $\hat{g}'_y(x_i)$ are respectively modified softmax outputs of the old copy of the network and of the current network, that differ from the original softmax operation due to the introduction of a "temperature" value T. This modified softmax is defined according to the following formulas for $\hat{q}_i'^y$ and $\hat{g}'_y(x_i)$:

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	**0.596**
Cross Entropy loss	0.83	0.717	0.653	0.598	0.558	0.545	0.502	0.477	0.459	0.447	0.578
MSE loss	0.856	0.693	0.628	0.588	0.564	0.55	0.5	0.467	0.45	0.43	0.573

Table 2. Test accuracies achieved by the baseline model and by the implemented versions of iCaRL with $CrossEntropy$ loss and MSE loss as classification loss at each incremental step (over the number of classes).

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	**0.596**
L_1 loss	0.869	0.643	0.615	0.587	0.576	0.546	0.51	0.478	0.45	0.445	0.572
MSE loss	0.879	0.721	0.664	0.634	0.591	0.554	0.506	0.475	0.444	0.435	0.59

Table 3. Test accuracies achieved by the baseline model and by the implemented versions of iCaRL with L_1 loss and MSE loss as distillation loss at each incremental step (over the number of classes).

$$\hat{q}_i'^y = \frac{\exp(\bar{a}_y(x_i)/T)}{\sum_j \exp(\bar{a}_j(x_i)/T)} \quad \hat{g}'_y(x_i) = \frac{\exp(a_y(x_i)/T)}{\sum_j \exp(a_j(x_i)/T)}$$

where $a_y(\cdot)$ is the logit associated to class y given by the FC output layer of the network being trained, and $\bar{a}_y(\cdot)$ is the same but given by the old configuration of the network (i.e. with pre-update parameters).

A temperature softmax accomplishes a similar operation to the standard softmax function, that is converting a set of logits to a probability distribution, but results in a more flattened and uniformly distributed result: higher values of T correspond to a softer, "smoother" probability distribution over classes. This is clearly observable from figure 10.

The outputs of our network are the activations of the FC output layer, therefore the temperature softmax can be used to emphasise the "hidden knowledge" of the model by highligthing the prediction probabilities that would have been otherwise flattened by a standard softmax operation. This allows the network to predict a class for a certain input, but also to state which classes are closest to the predicted one in probability terms.

As classification term, a standard cross entropy loss is used, which uses as target the ground truth labels y:

$$l_c(\Theta) = -\sum_{y=s}^{t} \delta_{y=y_i} \log \hat{g}_y(x_i)$$

where $\hat{g}_y(x_i)$ indicates the standard softmax output of a certain image x_i.

The overall loss, computed as the mean of the per-sample losses computed over training images, is:

$$L(\Theta) = \frac{1}{|D|} \sum_{(x_i,y_i)\in D} \omega_c l_c(\Theta) + \omega_d l_d(\Theta) \qquad (5)$$

where ω_d, ω_c are the weights for the classification and distillation term respectively. We finetune the hyperparameters associated to the losses and we obtain the best configuration at $T = 2, \omega_c = 1, \omega_d = 2$.

3.2.2 Experiments and Results

At the right temperature, the Temperature Softmax distillation, seems to be particularly effective in iCaRL's framework. Standard cross-entropy loss differently from binary cross-entropy loss works at lower value of learning rate, therefore we train iCaRL with temperature softmax distillation loss with an initial value for the learning rate equal to 0.2. The other hyperparameters are the same of iCaRL's implementation. This result does not come as a surprise, since the employed loss in this experiment is a slightly modified version of the original loss used by Li and Hoiem [11].

The only difference between our study and the one performed by Li and Hoiem is in the use of exemplars. This highlights the key role of this step in an incremental learning task. Indeed, the obtained results of the temperature distillation loss with the added exemplars, is very near to iCaRL's results. Nevertheless, the confusion matrix in fig. 11 shows that the temperature softmax framework seems to be more unbalanced towards new classes in the long run in comparison to iCaRL (fig. 5). The higher the temperature, the flatter the softmax probabilities becomes. With uniformly distributed softmax probabilities over classes, the network takes less information during the training process, and fails

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	**0.596**
T=2	0.84	0.729	0.645	0.595	0.561	0.538	0.503	0.472	0.453	0.436	0.577
T=5	0.824	0.71	0.646	0.595	0.58	0.545	0.5	0.465	0.455	0.432	0.575
T=10	0.799	0.692	0.617	0.574	0.543	0.518	0.484	0.462	0.443	0.418	0.555

Table 4. Test accuracies achieved by the baseline model and by the implemented versions of iCaRL with Temperature Distillation Loss at each incremental step (over the number of classes).

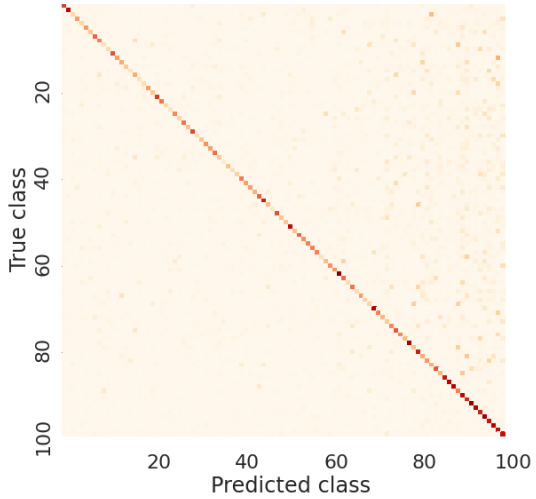

Figure 7. Confusion Matrix of iCaRL with L_1 distillation loss.

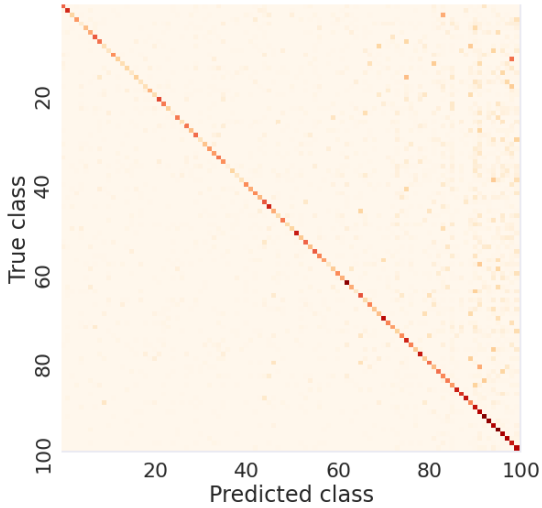

Figure 8. Confusion Matrix of iCaRL with MSE distillation loss.

more frequently in distinguishing classes. These assumptions on the temperature are confirmed by the results obtained, indeed tab. 4 proves that the higher the temperature value is, the lower are the performances at each incremental step.

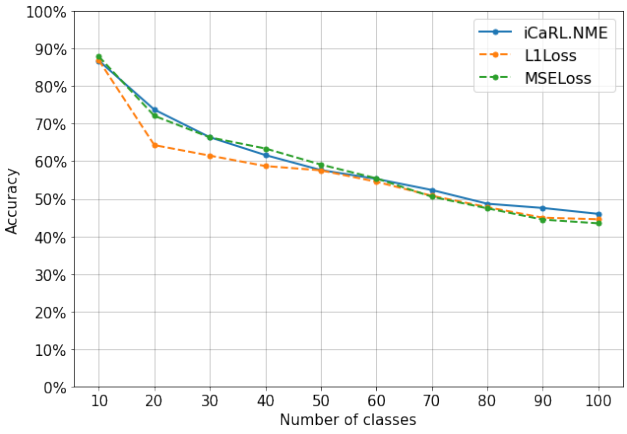

Figure 9. Evolution of the test accuracies over the number of classes incrementally seen by iCaRL and with different distillation losses (L_1 loss, MSE Loss).

Figure 10. Probabilities distributions of the same image at different temperature.

4. Ablation study on classifiers

4.1. Cosine normalization layer

4.1.1 Theoretical description

One of the main difficulties in the incremental learning setting is the imbalance between training images of the newly seen classes and stored exemplars of past classes during an incremental training step, and this problem leads to catastrophic forgetting, as Hou *et al*. have discussed in [5].

They describe a number of adverse effects which derive

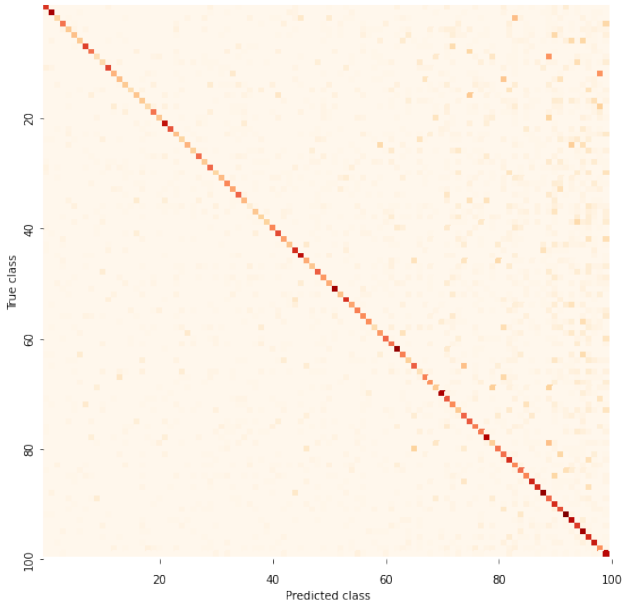

Figure 11. Confusion Matrix of iCaRL with Temperature Distillation Loss at T=2.

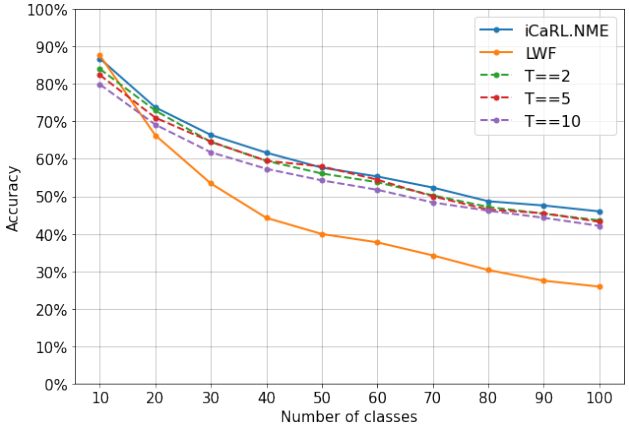

Figure 12. Evolution of the test accuracies over the number of classes incrementally seen by iCaRL, LWF and iCaRL with Temperature Distillation loss at different temperature values.

from this imbalance:

1. Imbalanced weight magnitudes: the magnitudes of the FC output layer's weight vectors associated to new classes are remarkably higher than those of old classes;

2. Deviation of features from weights: the previous knowledge, *i.e.* the relationship between the feature vectors of images of old classes and the weight vectors of old classes, are not well preserved;

3. Ambiguities between weights: the weight vectors of new classes are close to those of old classes in the weight vector space, often leading to ambiguities.

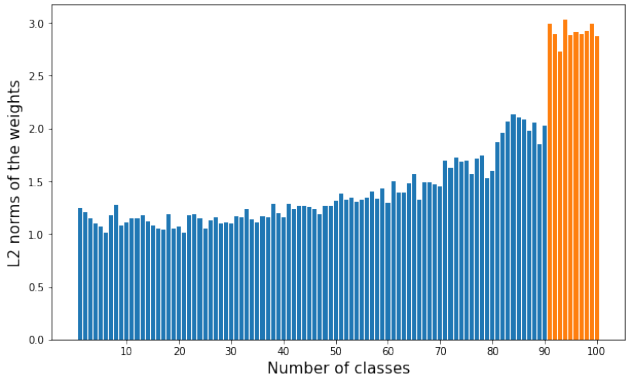

Figure 13. Visualization of the weights in the last FC layer for old classes (blue) and new classes (orange), after the 10 steps of 10-batch incremental training on CIFAR-100

The strategy used by [5] to tackle catastrophic forgetting is based on mitigating each of these adverse effects caused by the imbalance, by introducing three different components:

1. an output *cosine normalization layer* instead of a fully connected one, to tackle imbalanced weight magnitudes;

2. a *less-forget constraint* loss to tackle deviation of features from weights;

3. an *inter-class separation* loss to tackle ambiguities between weights

Each of these components will be discussed hereafter.

4.1.2 Cosine Normalization Layer

This approach is motivated by the fact that, according to [5], the last fully connected layer of the model will often learn weight vectors of higher norms for the most recent classes, if compared to those of the old classes; similarly, biases associated to new classes will tend to be higher. We verify the former claim by inspecting the weights of the last FC layer of the network after running a full 10-batch training on CIFAR-100 in the standard iCaRL setting. More specifically, we plot the L^2-norm of the weight vectors associated to each output neuron of the FC layer (see fig. 13). Indeed, the last 10 classes seem to consistently have vectors of higher norms.

We now attempt to verify if a normalization of the output layer weights is beneficial for the incremental learning of the model. To do so, we replace the usual fully connected layer at the end of the network with a different classifier: the cosine normalization (CN) layer proposed in [5]. Given an input image x, in a typical CNN the predicted probability of x belonging to class i is computed as follows:

$$p_i(x) = \frac{\exp(w_i^T \varphi(x) + b_i)}{\sum_j \exp(w_j^T \varphi(x) + b_j)}$$

where φ is the feature extractor, w and b are the weights and the bias vectors in the last FC layer. This is basically a FC layer followed by a softmax operation. Substituting the FC layer with a CN layer, we obtain:

$$p_i(x) = \frac{\exp(\eta \langle \overline{w}_i \, , \, \overline{\varphi}(x) \rangle)}{\sum_j \exp(\eta \langle \overline{w}_j \, , \, \overline{\varphi}(x) \rangle)}$$

where $\overline{v} = v / \|v\|_2$ denotes the L^2-normalized vector, and $\langle \overline{v}_1 \, , \, \overline{v}_2 \rangle = \overline{v}_1^T \overline{v}_2$ measures the cosine similarity between two normalized vectors. The learnable scalar η is introduced to control the peakiness of the softmax distribution since the range of $\langle \overline{v}_1 \, , \, \overline{v}_2 \rangle$ is restricted to $[-1, 1]$. Notice that, with this classifier, we normalized both weights of the CN layer and feature vectors. This method does not take biases into account: this way, we simultaneously tackle both problems stemming from imbalanced magnitudes.

4.1.3 Loss function

We combine the action of this new classification layer with:

- a standard cross-entropy loss as classification loss:

$$L_{\text{ce}} = -\sum_{i=1}^{|\mathcal{C}|} y_i \log(p_i)$$

  where $\mathcal{C}$ is the set of all the observed classes so far, y is the one-hot ground-truth label and p is the corresponding class probabilities obtained by softmax;

- the less forget constraint proposed in [5] as distillation loss, specifically designed for this classifier;

- the inter-class separation constraint, again proposed in [5]

The less-forget constraint encourages the new network to produce feature vectors similar to those that would have been produced by the old network (before the new classes were seen) by exploiting the fact that a cosine similarity between two normalized vectors is at most 1. We introduce a contribution to the total loss which has the form of a Cosine Embedding loss:

$$L_{\text{dis}}^G = 1 - \langle \overline{f}^*(x) \, , \, \overline{f}(x) \rangle$$

where $\overline{f}^*(x)$ is the feature mapping of the old network. The rationale behind this design is that the spatial configuration of the weight vectors of the seen classes, to a certain extent, reflects the inherent relationship among classes. Hence, to preserve the previous knowledge, a natural idea is to keep this configuration of weight vectors fixed (in fact, only feature vectors are considered which are not computed using the weight vectors, so minimizing this loss doesn't modify the weight vectors). With the weight vectors of old classes fixed, it is reasonable to encourage the feature vectors to be similar as in $L_{\textbf{dis}}^G$.

The inter-class separation constraint encourages separation between feature and weight vectors of different classes, by introducing a margin ranking loss. Specifically, for each stored exemplar x it attempts to separate the ground-truth old class (*i. e.* the class of x) from all the new classes by a margin m, using x itself as an anchor. We consider the weight vector associated to the ground-truth class as positive. To find the *hard* negatives, the K classes yielding the highest response to x are found, and their corresponding weight vectors are used as negatives for the corresponding anchor. The resulting margin ranking loss is:

$$L_{\text{mr}}(x) = \sum_{k=1}^{K} \max(m - \langle \overline{w}(x) \, , \, \overline{f}(x) \rangle + \langle \overline{w}^k \, , \, \overline{f}(x) \rangle, 0)$$

where m is the margin threshold, $\overline{w}(x)$ is the weight vector of the ground-truth class, $\overline{w}^k$ is one of the weight vectors associated to the top-K negative classes. K and m are hyperparameters.

It is worth noting that, the positive and the negatives for each anchor are weight vectors, instead of images (used in a typical margin ranking loss).

Combining the three losses, we reach a total loss given as:

$$L = \frac{1}{|\mathcal{N}|} \sum_{x \in \mathcal{N}} (L_{\text{ce}}(x) + \lambda L_{\text{dis}}^G(x)) + \frac{1}{|\mathcal{N}_{\text{o}}|} \sum_{x \in \mathcal{N}_{\text{o}}} L_{\text{mr}}(x) \quad (6)$$

where $\mathcal{N}$ is a training batch drawn from $\mathcal{X}$, $\mathcal{N}_{\text{o}} \subset \mathcal{N}$ is the set of stored exemplars contained in $\mathcal{N}$. λ is a loss weight, which is set adaptively according to the following equation:

$$\lambda = \lambda_{\text{base}} \sqrt{\frac{|\mathcal{C}_{\text{o}}|}{|\mathcal{C}_{\text{n}}|}}$$

where $|\mathcal{C}_{\text{o}}|$ and $|\mathcal{C}_{\text{n}}|$ are the number of old and new classes in each phase, λ_{base} is a hyperparameter. In general, λ increases as new classes are seen, reflecting the fact that degree of need of preserving the past knowledge is increasingly important (because stored exemplars of each old class become less and less, so the training dataset augmented with the exemplars becomes increasingly imbalanced).

4.1.4 Experiments and results

We evaluate the performances of the cosine normalization layer built upon the same ResNet-32 used in the iCaRL baseline model (section 2.1). Specifically, we compare the evolution of the test accuracy on CIFAR-100 between iCaRL.NME (Section 2.4) and iCaRL with cosine normalization layer as classifier and loss defined in Eq. (6) (iCaRL.CNL).

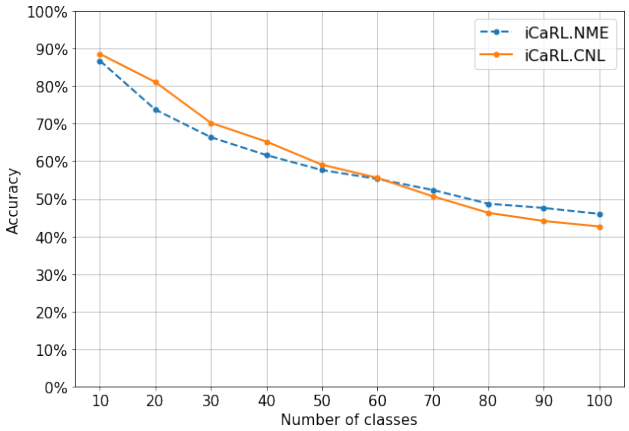

Figure 14. Evolution of the test accuracies of iCaRL with Cosine Normalization Layer and comparison with iCaRL with Nearest-mean-of-exemplars classifier

For training the latter, we chose the following hyperparameters: learning rate starts at 0.1 and is divided by 10 after 80 and 120 epochs (160 epochs in total); the network is trained by SGD with batch size 128 and L^2 weight regularization with weigh decay parameter of 5×10^{-4}; at most 2000 stored exemplars, $K = 2$, $m = 0.5$, $\lambda_{\text{base}} = 5$, as done in [5].

We report the evolution of the test accuracy in tab. 5 and we plot it in fig. 14.

The results show that using a cosine normalization layer along with its custom loss does achieve better results only in the first half of the incremental steps; the original iCaRL.NME still outperforms this new classifier in the long run. Despite having a worse final test accuracy, iCaRL.CNL still achieves a slightly better average incremental accuracy than iCaRL.NME. To better visualize how iCaRL.CNL behaves at tackling catastrophic forgetting, we can plot the confusion matrix associated to the final evaluation of iCaRL.CNL (on the whole test set). This is reported in the heatmap of fig. 15.

The confusion matrix clearly shows signs of (mild) catastrophic forgetting: images tend to be classified as one of the latest 10 classes seen during the incremental training.

Overall, we argue that the iCaRL.NME framework outperforms its variation iCaRL.CNL in the long run, and can manage the intrinsic imbalance of old classes data vs new classes data better. However, iCaRL.CNL still achieved better performances than iCaRL.NME on the first half of the incremental training steps. This may be a hint of the cosine normalization layer as classifier (and the new loss introduced) being better than the nearest-mean-of-exemplars classifier when the number of incremental training steps is limited; indeed, that was the particular setting in which Hou *et al*. [5] conducted their experiments (finetuning iCaRL.CNL initially with 50 classes and then performing

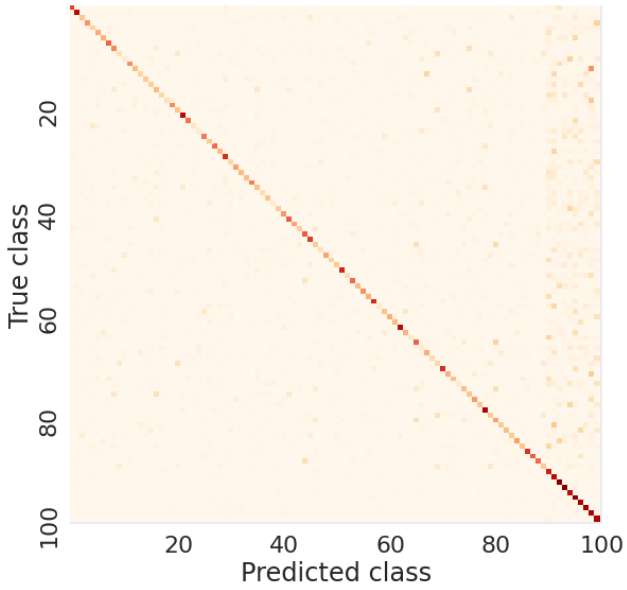

Figure 15. Confusion matrix of iCaRL.CNL

only 5 incremental steps with 10 new classes each, thus outperforming iCaRL.NME).

4.2. Bias Correction method

4.2.1 Theoretical Description

iCaRL model, due to the imbalance between the number of samples from the new classes and the number of exemplars from the old classes, has a bias towards the new classes. The hypothesis is that the last fully connected layer is biased as the weights are not shared across classes. To evaluate if the fully connected layer is heavily biased, we make two steps: (a) applying iCaRL model to learn both the features and fully connected layers, (b) freezing the feature layers and retrain the fully connected layer alone using all training samples from both old and new classes. Compared with iCaRL (figure 16) the accuracy of the final classifier on all the classes improves by 15%. These results validate the hypothesis that the FC layer is heavily biased.

The method introduced by Wu *et al*. in [15] is used in order to correct the bias introduced by the fully connected layer. This method includes two training stages:

1. We train the convolutional layers and the fully connected layer as in the iCaRL's framework.
2. We freeze both the convolutional and the fully connected layers, and we estimate two bias parameters by using a small validation set.

As loss for this method, we use the original loss function used by iCaRL (Section 2.3). The overview of this method can be seen at fig. 17. The final outputs of this network are the ones obtained from the BIC layer.

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	0.596
iCaRL.CNL	0.886	0.811	0.702	0.653	0.591	0.556	0.507	0.463	0.441	0.426	**0.603**

Table 5. Test accuracies comparison

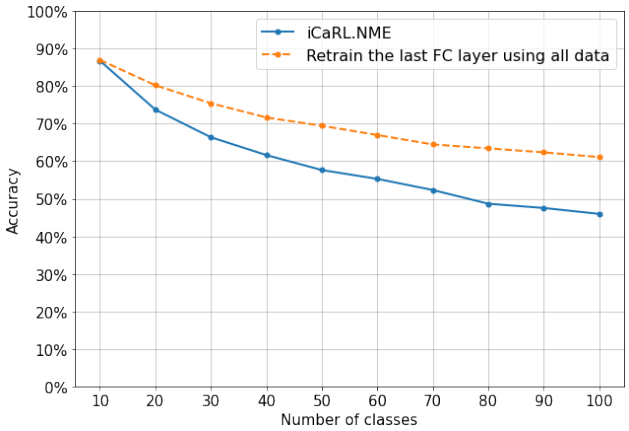

Figure 16. Experimental result to validate the bias in the last FC layer. Comparison of the obtained accuracies from iCaRL's model and the last FC layer retrained alone by using all training samples.

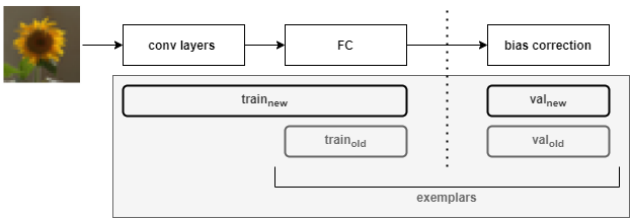

Figure 17. Overview of BIC method

Bias Correction layer Bias Correction layer is a linear model with two parameters α and β. This layer works as an identity for the output logits of the old classes $(1, ..., n)$ and we try to correct the bias on the output logits for the new classes $(n+1, ..., n+10)$:

$$q_k = \begin{cases} o_k & 1 \leq k \leq n, \\ \alpha o_k + \beta & n+1 \leq k \leq n+10 \end{cases}$$

where α and β are the bias parameters, and o_k is the output logit for the $k-th$ class. The bias parameters are shared by all new classes, and this allows us to estimate them with a small validation set. The classification loss used to optimize the bias parameters is:

$$L_b = -\sum_{y=1}^{n+10} \delta_{y=y_i} \log[\hat{q}_y]$$

where $\hat{q}_y$ is the softmax output of the y class.

Validation set The bias is estimated by using a small validation set. The samples for the new classes are split in a training set $train_{new}$ and a validation set val_{new}, and the exemplars for the old classes are split in a training set $train_{old}$ and in a validation set val_{old}. The dataset obtained by merging $train_{old}$ and $train_{new}$ sets is used to learn the convolutional layers and fully connected layers, while instead the obtained dataset from the merging of val_{old} and val_{new} sets is used in the bias correction layer as validation set. It is important to remark that val_{old} and val_{new} sets are balanced, namely the number of samples per class is the same in both the splits.

4.2.2 Experiments and Results

We evaluate the BIC Correction method by comparing the obtained results with iCaRL.NME and iCaRL.CNN (Section 2.4), which has a stronger relationship with this classifier.

For the training of iCaRL.BIC, the best found hyperparameters are the following: learning rate starts at 0.2 as in the case of the best obtained configuration of network with the use of original Binary Cross Entropy, as defined in iCaRL. The other hyperparameters, are the same used in iCaRL's implementation (section 2.4). Furthermore, the same hyperparameters are used for BIC layer, with the only difference that for this layer is used a different loss, that is standard CrossEntropy. Since only a few exemplars are stored for the old classes, it is critical to find a good split which deals with the trade-off between training the feature representation and correcting the bias in the FC layer. The best results are obtained with split 9:1 of $train_{old} : val_{old}$, as it is possible to verify from table 6

BIC method, with all tried splits, achieves better results in all the steps with respect to iCaRL.CNN, while instead, if we compare BIC method with its best split with iCaRL.NME, it attains greater performances in the first half of incremental steps, and slightly decrease in the second half, in comparison to iCaRL.NME, because it fails in distinguishing different classes as the overall number of classes increases.

4.3. K-Nearest Neighbors

In this section, we propose a variation of the iCaRL model that employs the k-Nearest Neighbor (commonly re-

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	**0.596**
iCaRL.CNN	0.861	0.712	0.642	0.585	0.542	0.499	0.470	0.430	0.412	0.386	0.554
iCaRL.BIC (9:1)	0.858	0.738	0.696	0.625	0.595	0.542	0.494	0.457	0.422	0.405	0.583
iCaRL.BIC (8:2)	0.850	0.731	0.684	0.612	0.580	0.529	0.481	0.444	0.413	0.387	0.571
iCaRL.BIC (7:3)	0.854	0.723	0.679	0.603	0.567	0.519	0.467	0.432	0.409	0.378	0.561

Table 6. Test accuracies achieved by iCaRL.NME, iCaRL.CNN at each incremental step, in comparison with BIC method implementation, with different splits of $train_{old} : val_{old}$.

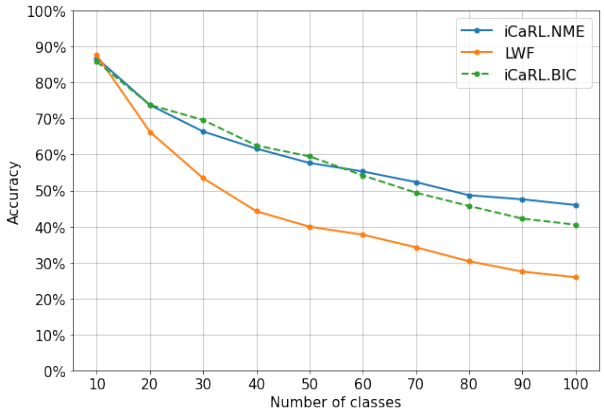

Figure 18. Evolution of the test accuracies of iCaRL with Nearest-mean-of-exemplars classifier, LWF and iCaRL with BIC layer.

ferred to as KNN), a well-known supervised learning algorithm to classify images. Neighbors-based classification is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the class which has the most representatives within the nearest neighbors of the point.

4.3.1 Theoretical description

We will first describe how the KNN algorithm works in general and then how it can be applied to the problem of image classification. Given a dataset composed of training points x, each with label y, we want to classify a query point $\tilde{x}$. To do so:

1. we establish a parameter K and a distance metric;
2. we find the K nearest training points from $\tilde{x}$ according to the chosen distance metric;
3. $\tilde{x}$ will be assigned the label that most frequently appears among its K nearest neighbors, in a *hard major voting* fashion.

A possible variation of the classic KNN algorithm just described is the *weighted KNN*, in which a weighted voting policy takes the place of the hard major voting policy: each neighbor point contributes to the final vote proportionally to the inverse of their distance from $\tilde{x}$; in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

To know the K closest elements to $\tilde{x}$, we need to compute the distance from $\tilde{x}$ to each training point. The most commonly chosen distance metric is the euclidean distance, which we also employ in our experiments. The main features of the KNN algorithm are its "automatic" non-linearity and its intrinsically incremental nature.

The latter aspect is surely the most interesting in our setting: indeed, as explained in section 2.4, iCaRL's NME classifier can be seen as a KNN classifier with K=1 in the feature space defined by φ. We shall too follow this approach: given a query image $\tilde{x}$ and the dataset composed of all the stored exemplars, we compute the feature vector $\varphi(\tilde{x})$ and the feature vectors of the exemplars and apply the KNN algorithm.

On the other hand, the downsides of this algorithm are the amount of data which we need to store (normally, the entirety of the training set) and its computational cost at classification time ($O(n)$, where n is the total number of training points).

The choice of the hyperparameter K is of critical importance, and will considerably influence the model's performance. If K is too small, we may consider too few points for the classification and make the classifier sensitive to noise (overfitting may occur). If K is too large, the classifier may take into account too many images across different class regions and consequently underfit.

The interest in this classification approach stems from the fact that objects of the same class are of course closely mapped in the feature space and, often, class clusters appear well separated from one another. In this scenario, a KNN could operate well; in our experiments, we shall compare iCaRL.KNN to iCaRL.NME in terms of incremental test accuracies.

4.3.2 Experiments and results

We experiment on the iCaRL framework by substituting the original NME classifier with a KNN classifier in the feature space φ. Incremental training is carried out as in standard iCaRL, and most of the other aspects of the framework are left untouched. After updating the feature representation for an incoming batch of classes, we fit the KNN on top of the architecture using the feature vectors of the currently available exemplars. We keep the same herding strategy proposed in [13]. This results in exemplars of the same class being more concentrated around a single point (the class mean), which is clearly beneficial to the KNN algorithm. To compute the distance between a query image $\tilde{x}$ and each stored exemplar $x^{(k)}$ in the feature space φ, we use the euclidean distance:

$$d(\,\varphi(\tilde{x})\,,\,\varphi(x^{(k)})\,) = \sqrt{\sum_{j=1}^{64} \left(\varphi(\tilde{x})_j - \varphi(x^{(k)})_j\right)^2}$$

Note that $\varphi(\cdot)$ is 64-dimensional in our setting.

As for the hyperparameter K, we experiment with two different approaches:

- keeping K fixed at all times;
- adaptively change K according to the current size m of the exemplar sets.

Moreover, we experiment both with classic KNN and weighted KNN (wKNN).

In table 7 we report the evolution of the overall test accuracy as well as of the test accuracy on old classes and new classes, at each incremental step, for different values of K and the two different voting policies.

Note that by setting $K = (3/2)m$ we intentionally take into account more points than those contained in a single exemplar set. Such an attempt is motivated by the assumption that fitting the whole appropriate exemplar set (and $0.5m$ points of incorrect exemplar sets) within the neighbor set of the query point would ensure a correct classification.

KNN doesn't outperform the NME classifier, but it gets really close to NME's results by setting $K = (3/2)m$, either with hard major voting or weighted voting scheme. In general, adaptively changing K has proven better than keeping it fixed to a small value like 1 or 5, in terms of both final and avg. incremental test accuracy. This result is expected, as smaller values of K will make the model less robust against noise/outliers. Two ways to smooth out the impact of isolated noisy training points are increasing the value of K (even though this increases the computational cost of classification) and employing a distance-weighted voting policy: the results obtained seem to reflect this theoretical overview.

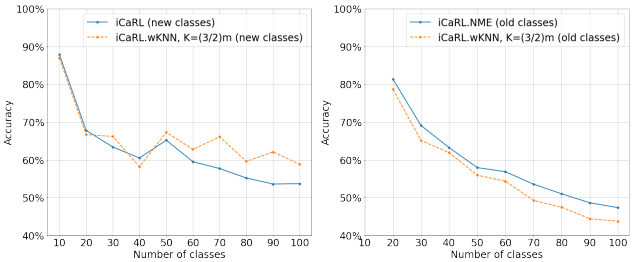

Figure 19. Evolution of the test accuracies of iCaRL.wKNN with $K = (3/2)m$ for old classes and new classes, and comparison with iCaRL.NME

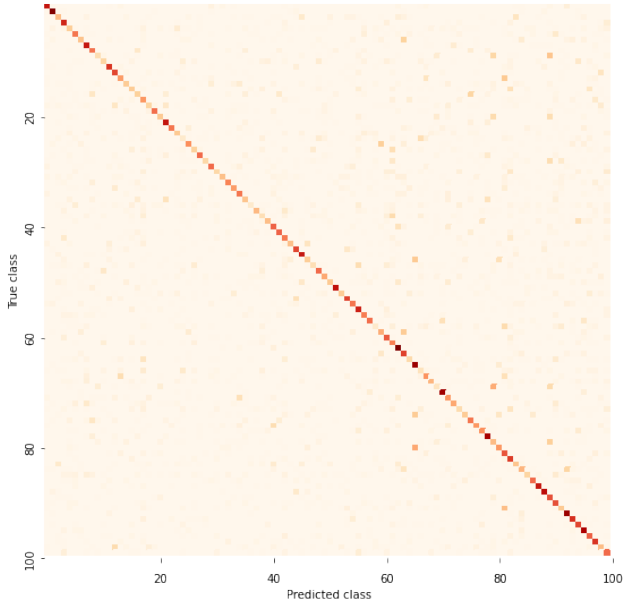

Figure 20. Confusion matrix of iCaRL.wKNN with $K = (3/2)m$

To better compare KNN and NME classifiers, in fig. 19 we plot the evolution of the test accuracy on old classes and new classes for iCaRL.NME and iCaRL.wKNN with $K = (3/2)m$. We can see that using the NME classifier ensures a better test accuracy on old classes, in all the incremental steps, although the difference is not large. This can also be seen in the confusion matrix in fig. 20. As KNN seems to be more biased towards new classes, thus suffering from a slight catastrophic forgetting phenomenon, we conclude that it does not outperform the NME classifier under any aspect.

4.4. SVM

4.4.1 Theoretical Description

Support-Vector Machines are supervised models with associated learning algorithms that analyze data for classification and regression tasks. A support-vector machine algorithm constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used in

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	**0.596**
iCaRL.KNN, K=1	0.868	0.731	0.654	0.591	0.560	0.522	0.476	0.449	0.421	0.407	0.568
iCaRL.KNN, K=5	0.887	0.741	0.651	0.609	0.582	0.549	0.505	0.470	0.450	0.425	0.587
iCaRL.KNN, K=(1/2)m	0.876	0.724	0.667	0.625	0.588	0.553	0.511	0.483	0.455	0.434	0.591
iCaRL.KNN, K=(3/2)m	0.890	0.756	0.675	0.609	0.580	0.546	0.518	0.482	0.457	0.451	**0.596**
iCaRL.wKNN, K=1	0.873	0.731	0.658	0.598	0.562	0.522	0.486	0.447	0.429	0.409	0.572
iCaRL.wKNN, K=5	0.871	0.750	0.664	0.617	0.591	0.560	0.517	0.486	0.461	0.443	**0.596**
iCaRL.wKNN, K=(1/2)m	0.856	0.741	0.677	0.607	0.580	0.546	0.504	0.478	0.463	0.443	0.589
iCaRL.wKNN, K=(3/2)m	0.870	0.727	0.655	0.610	0.582	0.558	0.517	0.490	0.464	0.453	0.592

Table 7. KNN. Test accuracies comparison

learning algorithms. Intuitively, a good separation of data points is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class (so-called "minimum geometric margin") since in general the larger the margin, the higher the generalization of the classifier. In our specific case is used a multiclass version of the SVM. The multiclass problem is broken down to multiple binary classification cases, which is also called one-vs-one SVM.

Soft-Margin Linear SVM The searched hyperplane in the Linear SVM algorithm is of the form $\langle \vec{\omega}, x \rangle + b = 0$ where $\vec{\omega}, b$ are defined in order to maximize the distance of this plane to the nearest point from either group. The Hard Margin formulation of this problem is:

$$\arg\min_{(\omega,b)} \frac{1}{2} ||\omega||^2$$

s.t. $y_i(\langle(\omega, x_i\rangle + b) > 1$.

This formulation refers to the case in which data are linearly separable. The Soft-Margin approach overcomes this problem, by introducing a slack variable ξ for each sample. The optimization problem can be rewritten as

$$\arg\min_{(\omega,b)} \frac{1}{2} ||\omega||^2 + C \sum_i \xi_i$$

s.t. $y_i(\langle \omega, x_i \rangle + b) > 1 - \xi_i$ and $\xi_i \geq 0$.

Slack variables are introduced in order to allow classification mistakes. The hyperparameter C determines the trade-off between increasing the margin size with these variables and ensuring that data lie on the correct side of the margin.

RBF SVM Another approach to overcome the problem of non-linearly separable data is to introduce a kernel function, which enables to operate in a high-dimensional, implicit feature space, without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all data pairs of data in the feature space.

$$K(x, x') = \langle \psi(x), \psi(x') \rangle$$

Specifically, the RBF kernel formula is:

$$K(x, x') = e^{-\gamma ||x - x'||^2}$$

4.4.2 Experiments and Results

In this section iCaRL's NME classifier is substituted with a Support Vector Machine in the feature space φ. The SVM classifier is fitted on top of the architecture with the feature vectors of the currently available exemplars, while instead the other used approaches in the original iCaRL's framework are left unchanged. The two studied implementations of iCaRL.SVM in this section are:

- *iCaRL.Linear_SVM*: iCaRL implementation with Linear SVM as classifier;
- *iCaRL.RBF_SVM*: iCaRL implementation with RBF SVM as classifier;

Among the different hyperparameter configurations tested, the most relevant ones in terms of test accuracies are reported in Table 8. By analyzing fig. 21, it is evident that iCaRL.RBF_SVM is consistently accurate on new classes, while it tends to forget old classes more quickly than iCaRL. This can be demonstrated by computing the overall silhouette score of the clusters belonging to the different classes. The 100 exemplar sets achieve a silhouette score of 0.06. Since value of silhouette score near to 0 means that the clusters in the feature space are not well separated, it seems that the network is not very good in keeping exemplars together in the feature space. Indeed, SVM with RBF kernel which

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	**0.596**
iCaRL.Linear_SVM(C=1)	0.856	0.729	0.656	0.602	0.571	0.551	0.516	0.486	0.467	0.445	0.587
iCaRL.RBF_SVM(C=1)	0.861	0.74	0.656	0.603	0.567	0.541	0.515	0.483	0.463	0.451	0.588
iCaRL.Linear_SVM(C=10)	0.862	0.727	0.652	0.594	0.572	0.541	0.504	0.47	0.452	0.439	0.581
iCaRL.RBF_SVM(C=10)	0.86	0.724	0.655	0.606	0.58	0.546	0.509	0.484	0.46	0.44	0.586

Table 8. SVM. Test accuracies comparison

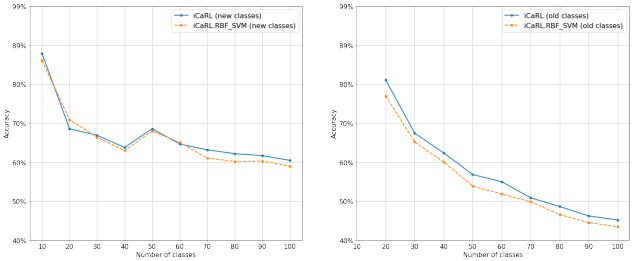

Figure 21. Evolution of the test accuracies of iCaRL and iCaRL.RBF_SVM on old and new classes separately

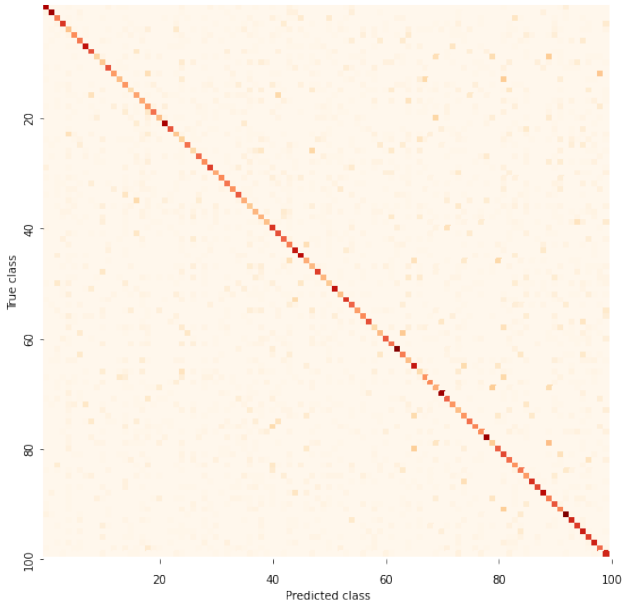

Figure 22. Confusion Matrix of iCaRL.RBF_SVM with $C = 1$.

enable to operate in a high-dimensional space, tends to work better than Linear SVM, although it is not capable of reaching iCaRL's performances.

5. Variations

The major challenge in incremental learning is the assumption that past data cannot be fully retained for future incremental training steps: as we have discussed, the imbalance between previous and new data used at training time in an incremental learning step seems to be the crucial cause of the catastrophic forgetting phenomenon, since it leads model to be biased towards new classes. [5]

The iCaRL method achieved quite good performances on the CIFAR-100 dataset, scoring a final test accuracy of 0.46 after training for 10 incremental steps (10 batches of 10 classes each), proving effective in tackling the catastrophic forgetting phenomenon. As we have seen, iCaRL employs herding-based prototype rehearsal and knowledge distillation to tackle the imbalance issue, therefore addressing catastrophic forgetting to a certain extent. However, since only a limited number of exemplars for each seen class can be stored in memory, the imbalance between training images of new classes and stored exemplars of old classes seems to be a problem with no real solution whatsoever.

However, more can be done. Several strategies exist to further tackle the imbalance between minority and majority classes at training time, which were not explored by the original iCaRL framework: among them, *image augmentation* and *oversampling* techniques.

Image augmentation The field of *image augmentation* encompasses a suite of techniques used to enhance the (virtual) size and quality of a training dataset such that the model can more efficiently learn to extract general features from the training images, instead of adapting to noise present in data. [14]

Geometric transformations (such as random horizontal/vertical flip, random rotations, shear, padding and many others) and color transformations (blur, saturation, color jitter and so on) can be applied to training images before being used to train an image classification model such as a CNN. A key concept of data augmentation is that the deformations applied to the labeled data do not change the semantic meaning of the labels. Taking an example from computer vision - our setting - a rotated, translated, mirrored or scaled image of a sunflower would still be a coherent image of a sunflower, and thus it is possible to apply these deformations to produce additional training data while preserving the label of the image (indeed, we refer to these operations

as being *label preserving*).

By training the network on the additional, augmented data, the hope is that the network becomes *invariant* to these deformations and generalizes better to unseen data. In the context of deep learning, we may say that a CNN that is capable of generalizing well to new, unseen data is one whose learned convolutional kernels are able to extract very general features from training data, such as low-level shapes and color patterns which are obviously invariant to geometric and color transformations.

These features might be shared by many different (and visually diverse) classes. Therefore we argue that an incremental learning model will be more easily trained on new unseen classes when it has learned to extract general features from the previously seen classes. In the end, we expect that this will at least partially make up for the inevitable imbalance between new and old data, intrinsic in incremental learning.

In addition to augmentation techniques on training data, we also experimented with test-time image augmentation. Many research reports have shown the effectiveness of augmenting data at test-time as well [14] [10] [3]. This can be seen as analogous to ensemble learning techniques in the data space: by taking a test image and augmenting it in the same way as the training images, a more robust and confident prediction can be derived. This comes at a computational cost depending on the augmentations performed.

Oversampling techniques Usually, when using image augmentation operations on training images, they are applied on the fly, hence not having to store the augmented images in memory. Therefore, we can clearly say that the size of the training set only increases *virtually*. As the training of a model is usually done for multiple epochs (i.e. multiple full passes on the training set), and since image augmentations are usually random (the same image is statistically never augmented in the same exact way twice), this does not constitute a problem: employing image augmentation is virtually like training with a way bigger "augmented" training set, from which only a subset of images is used in each epoch.

However, in the incremental learning setting, we are still interested in "concretely" solving the fundamental problem of imbalance between old and new classes at training time. In our setting, at any incremental training step i, the number of images seen by the model at training time is equal to $500 \times 10 + K$ where K, the number of stored exemplars is fixed and set to 2000. We could completely solve the imbalance problem by a trivial solution: oversampling the m exemplars of each of the old classes, so to have 500 images per seen class. This may be dangerous, for two main reasons:

- the number m of exemplars for each class decreases with time; at later incremental learning steps this means that m is quite small. Oversampling m exemplars to obtain 500 images per class is extreme and hazardous in these cases, since we may very easily incur in overfitting problems (even when image augmentation is applied to each oversampled image) [8];

- "full" oversampling of the exemplars would mean linearly increasing the amount of training images at each consecutive incremental training step: this is computationally unfeasible on the long run (at each consecutive incremental learning step, the training set would be 5000 images larger than the previous step).

Our proposed usage is performing a partial random oversampling of the exemplars, so that they are brought from $K = 2000$ to a fixed $K' > K$, independently from the incremental training step we are at and starting from the second incremental step (i.e. when exemplars begin to being used). The choice of $K' > K$ is an hyperparameter, which should be high enough to address imbalance between minority and majority classes effectively (although partially), but low enough to keep the computational cost of training low and avoid overfitting issues. We arbitrarily chose a value of $K' = 5000$; on average, each old class is represented by $(5/2)m$ exemplars at training time. Image augmentation operations are applied on the whole resulting training set, composed of the 5000 images of new classes plus the 5000 oversampled exemplars of old classes.

We also experimented with SMOTE (Section 5.2), an oversampling method which generate new synthetic data by averaging images belonging to the same class in all pixels .

In the next sections, several image augmentation and oversampling techniques will be discussed and tried out, to examine how applying these strategies affects the overall performances of iCaRL.

5.1. TenCrop

In this section we describe TenCrop, a test-time image augmentation technique widely employed in the deep learning community [10], [3].

5.1.1 Theoretical description

Applying TenCrop to a test image $\mathbf{X}$ of size $N \times N$ and classifying it simply consists in the following steps:

1. choose $N' \leq N$ and extract five $N' \times N'$ crops from $\mathbf{X}$ (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all);

2. predict the label of each of the ten crops with a classifier of choice;

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	0.460	0.596
w/ TenCrop	0.875	0.759	0.689	0.647	0.606	0.571	0.524	0.504	0.482	**0.465**	**0.612**

Table 9. Test accuracies comparison, with and without TenCrop

Figure 23. TenCrop transformation on an image of CIFAR-100.

3. average the ten predictions with a voting policy (e.g. hard major voting); X will be assigned the most voted label.

We stress the fact that $N' \leq N$ causes the model to classify images of smaller dimensions. In order to obtain $N \times N$ crops from the original image $\mathbf{X}$ we can for example pad in advance the original image at the four borders with black pixels and extract $N \times N$ crops from this new enlarged image. This is the approach used in our experiment. In fig. 23 we show a CIFAR-100 image of a sunflower before and after the TenCrop transformation has been applied.

5.1.2 Experiments and results

We perform a full incremental training of iCaRL on CIFAR-100 and evaluate it after each incremental training step by augmenting test images with the TenCrop augmentation. Before taking the five crops, we pad the test images with strips of black pixels; the new images will be 40×40, and the crops 32×32.

In table 9 we report the evolution of the overall test accuracy, and compare it to the one obtained without TenCrop.

As we could expect, using TenCrop has proved beneficial to the performances of iCaRL.NME. In particular, the overall test accuracy after each incremental training step is always more than the one obtained without using TenCrop, meanwhile the average incremental accuracy has increased by a moderate amount (1.6%). For this reason, we decided to include TenCrop in other experiments concerning different image augmentation techniques in the remainder of this section.

5.2. SMOTE

5.2.1 Theoretical Description

In this section, we propose the SMOTE oversampling strategy [12] in which the minority classes are oversampled by generating "synthetic" samples exploiting existing ones.

Each minority class is oversampled by taking each of its examples and introducing synthetic examples along the line segments joining at random any of its k nearest neighbors belonging to the same class. The hyperparameter k (positive integer) controls the maximum amount of oversampling which can be obtained: if a minority class is represented by m examples, the maximum amount of synthetic examples that SMOTE can generate for that class is $k \cdot m$ (maximum oversampling of $100 \cdot k\%$).

Increasing k will generate more diverse samples. We will chose $k = 5$ for our experiment, since it is the default and most commonly chosen value for k. For instance, if the amount of over-sampling needed is 200%, only two neighbors are randomly chosen from the $k = 5$ nearest neighbors and one sample is generated in the direction of each; moreover, if the amount needed is 250%, an oversampling of 200% is first performed and then an additional 50% is performed by coupling 50% of the minority class samples (chosen at random) with one of their $k = 5$ nearest neigbors (chosen at random too), and finally generating a synthetic sample for each couple.

A synthetic sample x_{new} is generated with the following formula:

$$x_{\text{new}} = x_i + \lambda(x_{nn} - x_i)$$

where x_i is an existing sample of the minority class, x_{nn} is one of its k neighbors and λ is a random number between 0 and 1. This is basically a random convex combination of x_i and x_{nn}: this causes the selection of a random point along the line segment between the two samples x_i and x_{nn}.

In our setting, the samples x_i's are $3 \times 32 \times 32$ tensors, representing images. In fig. 24 an example of SMOTE applied on an image of class "sunflower" of CIFAR-100 is reported.

5.2.2 Experiments and results

The SMOTE technique is applied to address imbalance between old previously seen classes and current classes to learn. To address this problem, starting from the second incremental step 3000 new synthetic images are created by

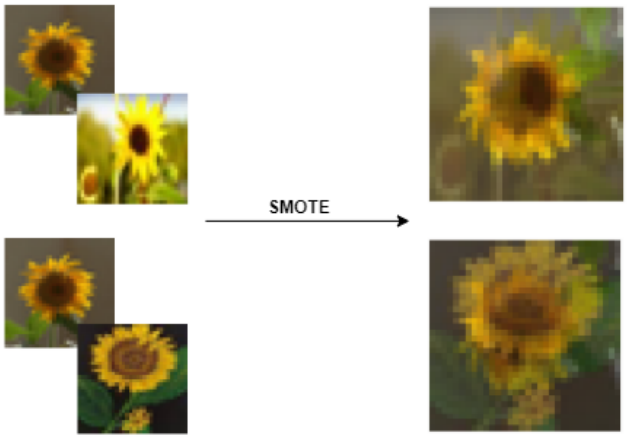

Figure 24. SMOTE applied on a class of CIFAR-100

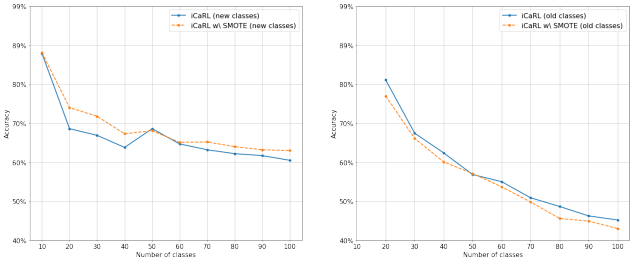

Figure 25. iCaRL's performance on old and new classes by applying SMOTE for rebalancing purposes compared with iCaRL's original performance.

SMOTE, from the original exemplars set, bringing the number of images of the old classes to 5000: each old class shall be represented by $(5/2)m$ exemplars instead of m (a 150% oversampling). SMOTE is purely based on the geometrical information retained at each training step within the exemplar sets, whose size decreases over time, therefore despite the increase of the number of exemplars of the old classes, we do not expect much improvement. From figure 25, it is visible that SMOTE lowers the performances on the old classes as the number of exemplars for each class decreases. Trying to re balance the data by solely exploiting the information that is encoded in exemplars' geometrical disposition is not enough. Indeed, despite on average SMOTE improves the performances of iCaRL, its performance are in general lowered as the number of stored exemplars per old class decreases, both with SMOTE alone and with SMOTE and TenCrop at test-time.

5.3. Sample Pairing

5.3.1 Theoretical Description

This section describes the SamplePairing data augmentation technique, firstly introduced by Inoue in [6]. The basic idea of this technique is to synthesize a new image from two images randomly picked from both the training images

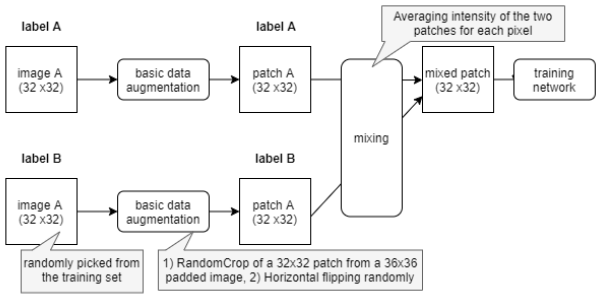

Figure 26. The used technique in Sample Pairing.

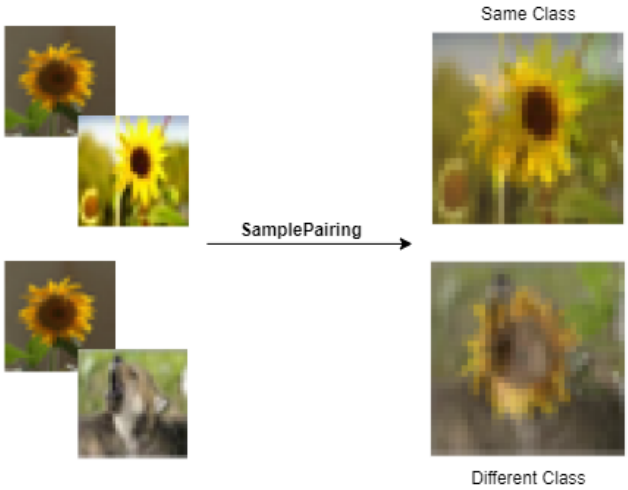

Figure 27. Sample Pairing technique applied on an image of CIFAR-100

of current classes and the stored exemplars of old classes. The new image is generated by averaging two images pixelwise. For each training epoch, all samples are fed into the network for training in randomized order. In this technique, another image is randomly picked from the training set, and an average of the two images is computed. The network is fed with the mixed image associated with the label of the first considered image. Due to the fact that the two images are equally weighted in the mixed image, it is therefore difficult for the classifier to correctly predict the label of the first image, unless the labels of the images from which it is taken the average, are the same. Other data augmentation techniques employed in addition to this technique (random crops, random horizontal flips, standardization), are applied before the two images are mixed with SamplePairing.

5.3.2 Experiments and results

A full incremental training of iCaRL on CIFAR-100 is performed, by appling Sample Pairing on all the training images, exemplars included. At first, some basic transformations (random crop, random horizontal flip, standardization)

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	0.596
w/ SMOTE	0.881	0.755	0.684	0.619	0.593	0.5556	0.52	0.479	0.465	0.444	0.599
w/ SMOTE + TenCrop	0.878	0.765	0.691	0.625	0.598	0.561	0.524	0.48	0.472	0.454	**0.6**

Table 10. Test accuracies comparison

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	0.460	0.596
w/ SamplePairing	0.79	0.709	0.665	0.619	0.595	0.571	0.534	0.508	0.475	0.466	0.593
w/ SamplePair. + TenCrop	0.801	0.712	0.664	0.623	0.602	0.58	0.542	0.512	0.489	**0.472**	**0.6**

Table 11. Test accuracies comparison

are applied on the training images, then with a probability of $p = 0.8$ each of these images is averaged pixel-wise with a randomly selected image taken from the training set, or it remains the same with a probability of $1 - p$. The probability $p = 0.8$ is chosen accordingly to the paper [6].

Furthermore, in order to slightly correct imbalance between old and new classes, new 3000 synthetic images are generated through random oversampling, prior to applying SamplePairing. Although iCaRL with SamplePairing shows a lower average incremental accuracy, the final test accuracy value on the 100 classes is greater than iCaRL's one. The application of TenCrop at test-time not only increase the average accuracy, but moreover also increases the final incremental step test accuracy.

5.4. PatchShuffle Regularization

5.4.1 Theoretical description

For image classification, when lacking sufficient training data, the learned model may be misled by the irrelevant local information which can be regarded as noise. Moreover, in most classification tasks, it is the overall structure rather than the detailed local pixels that has a large influence on the performance of a CNN model. The model should be able to identify the input image correctly if pixels within local structures change in a label-preserving way, i.e. in a way that does not destroy the overall view of an image. From another perspective, if we consider that human will probably not be confused about the the image content under moderate extent of local blur, it is expected that a data model such as a CNN should behave similarly.

To address this issue, Kang *et al.* [7] proposed PatchShuffle, an image augmentation technique which consists in blurring an image or feature map by partitioning it in rectangular patches and randomly shuffling pixels channel-wise. PatchShuffle is applied randomly with a fixed probability to training images and their feature maps.

Figure 28. Random shuffling pixels of an image's color channel or feature map in a PatchShuffle

Let us consider an image's color channel or feature map $\mathbf{X}$ of size $N \times N \times K$. Note that K represents the number of color channels for an image ($K = 3$ for a RGB image), meanwhile for a feature map K represents the number of kernels of the convolutional layer corresponding to that feature map. A random switch r controls whether $\mathbf{X}$ needs to be PatchShuffle'd; let p the probability that $r = 1$ (and so $r = 0$ with probability $1 - p$). Therefore, after the PatchShuffle routine, the resulting image or feature map can be represented as

$$\tilde{\mathbf{X}} = (1 - r)\mathbf{X} + rT(\mathbf{X})$$

where $T(\cdot)$ denotes the PatchShuffle transformation. To apply T to $\mathbf{X}$, we first split $\mathbf{X}$ into non-overlapping patches with sizes of $n \times n$ elements. Within each patch, the elements are randomly shuffled, as shown in fig. 28.

In fig. 29 we compare a 32×32 training image of CIFAR-100 depicting a sunflower and a PatchShuffle'd version of it. Note that in this image the same permutation of pixels is applied on each color channel. This is the way we

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	**0.460**	0.596
w/ PatchShuffle	0.873	0.723	0.666	0.607	0.585	0.546	0.512	0.483	0.460	0.450	0.590
w/ PatchShuffle + TenCrop	0.876	0.751	0.676	0.617	0.586	0.555	0.513	0.487	0.466	0.454	**0.597**

Table 12. Test accuracies achieved by using PatchShuffle at each incremental step (over the number of classes

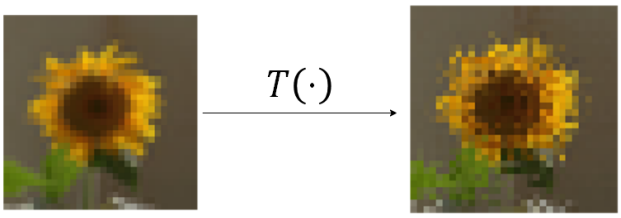

Figure 29. PatchShuffle transformation on an image of CIFAR-100; here, the same permutation of pixels is applied on each color channel.

will apply PatchShuffle on a raw image.

5.4.2 Experiments and results

We perform a full incremental training of iCaRL on CIFAR-100 by applying the PatchShuffle image augmentation on training images (exemplars included) and their feature maps, with shuffle probability $p = 0.05$ and patches of size 2×2. This choice of hyperparameters has been taken from [7], as in their experiments these two values let them achieve the lowest test error on CIFAR-10 (a variation of CIFAR-100 with only 10 classes [9]). After applying the PatchShuffle to a training image, the default iCaRL pre-processing operations are applied (random crop, random horizontal flip, standardization). Starting from the second incremental step, before applying the PatchShuffle and pre-processing operations, the $K = 2000$ stored exemplars of the old classes are brought to 5000 through random over-sampling, to address (at least partially) the imbalance problem between old and new classes at training-time.

We also repeated the same experiment, but this time using TenCrop at test-time. In table 12 we report the evolution of the overall test accuracy in the two experiments, and compare them to that of iCaRL.NME.

Apparently, PatchShuffle was not very effective in improving generalization on CIFAR-100. This may be caused by the fact that CIFAR-100 is a dataset composed of small, low-resolution images. As stated before, PatchShuffle should be a label-preserving operation: applying it to low-resolution images might make them unrecognizable not only to a machine learning model but also to a human observer, as it may blur too much some critical underlying features of the objects depicted in the images. Supplementing PatchShuffle with TenCrop improved performances by a slight amount, but only with respect to average incremental accuracy (which is in this case comparable to that of the original iCaRL).

5.5. AutoAugment

One of the downsides of data augmentation is that effective data augmentation policies must be manually found for a target dataset. Cubuk *et al*. [1] proposed a simple procedure called AutoAugment to automatically find good data augmentation policies from data alone.

5.5.1 Theoretical description

In [1] they formulate the problem of finding the best augmentation policy as a discrete search problem. AutoAugment consists of two components: a reinforcement learning algorithm to search over different augmentation policies, and a policy space $\mathcal{S}$ over which the algorithm finds the most effective policy(ies) for a certain dataset.

Policy space The policy space $\mathcal{S}$ is the set of all possible augmentation policies over which AutoAugment finds the best policies for a certain image dataset. Each policy $S \in \mathcal{S}$ consists of a set of 5 sub-policies with each sub-policy consisting of two image operations to be applied in sequence. Additionally, each operation is also associated with two hyperparameters:

1. the probability of applying the operation;
2. the magnitude of the operation.

AutoAugment searches over a total of 16 operations: ShearX/Y, TranslateX/Y, Rotate, AutoContrast, Invert, Equalize, Solarize, Posterize, Contrast, Color, Brightness, Sharpness, Cutout [2] (which is very similar to Random Erasing, discussed in section 5.6), Sample Pairing (discussed in section 5.3). Each of these operations comes with a default range of magnitudes, uniformly discretized into 10 values, so that a discrete search algorithm can be used to find them. Similarly, also the probability of applying each operation is uniformly discretized into 11 values (from 0.0 to 1.0). The default range of magnitude for each operation is reported in the appendix of [1].

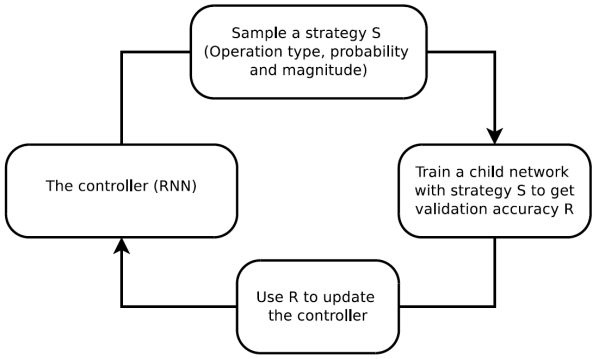

Figure 30. The policy learning algorithm used by AutoAugment.

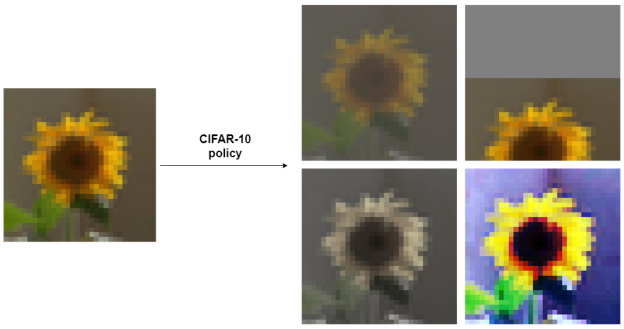

Figure 31. AutoAugment's CIFAR-10 policy applied on an image of CIFAR-100 (four times).

Policy search algorithm Since we didn't run AutoAugment from scratch on CIFAR-100, but rather used the best policies found for CIFAR-10 in [1] we will not go deep in the details of the policy search algorithm; we will hereafter describe it at a high level.

In fig. 30 we report an overview of the policy search algorithm employed by AutoAugment.

A controller recurrent neural network (RNN) predicts an augmentation policy S from the policy space $\mathcal{S}$. Each of the 5×2 operations of S comes with information on the probability of using that operation on an image, and the magnitude of that operation. The predicted policy S will be used to train a "child" neural network with a fixed architecture.

To train this child model, the training data of the target dataset is split in two subsets, such that a small validation set can be obtained. The child model is trained with augmented data generated by applying the 5 sub-policies of S on the training subset. For each example in a mini-batch, one of the 5 sub-policies is chosen randomly to augment the image. The child model is then evaluated on the validation set to measure the accuracy R, which is then used as the "reward signal" to train the recurrent network controller.

The reward R, which measures how good the policy is in improving the generalization of the child model, will be used with policy gradient methods to update the controller so that it can generate better policies over time.

To end this subsection on the theoretical aspects of the AutoAugment method, in fig. 31 we apply the policy found for CIFAR-10 (see next subsection) to a 32×32 training image of CIFAR-100 depicting a sunflower. Four such images are shown, for the sake of comparison.

5.5.2 Experiments and results

In [1] they used AutoAugment to find the best augmentation policies for CIFAR-10 and then they evaluate the found policies also on CIFAR-100, proving that AutoAugment policies found on a given dataset can be effectively transfered to visually similar datasets. Although CIFAR-10 has 50,000 training examples, they performed the search for the best policies on a "reduced" version of CIFAR-10 consisting of 4,000 randomly chosen examples, to save time for training child models during the augmentation search process.

The policies found during the search on reduced CIFAR-10 are then used to train final models on CIFAR-10 and CIFAR-100. They concatenated sub-policies from the best 5 policies found by the controller, to form a single policy $\mathcal{S}_{\text{CIFAR}}$ with 25 sub-policies. On CIFAR-10, AutoAugment picks mostly color-based transformations. For example, the most commonly picked transformations on CIFAR-10 are Equalize, AutoContrast, Color, and Brightness Geometric transformations are rarely found in good policies. This policy has been tested and shown in fig. 31. The policy found on CIFAR-10 is fully reported in the appendix of [1], as well as in our code.

For our experiment, we incrementally train the iCaRL model on CIFAR-100 augmenting the training images (exemplars included) with the AutoAugment policy of CIFAR-10. We implemented the same pre-processing steps used in [1]. For a given training image:

- the image is padded at the borders and a random 32×32 crop is taken;
- a random horizontal flip is applied with probability 0.5;
- apply one sub-policy chosen at random in $\mathcal{S}_{\text{CIFAR}}$;
- apply a 16×16 Cutout;
- standardize the resulting image by subtracting the per-channel means and dividing by the per-channel standard deviations of CIFAR-100.

To address imbalance between old classes and new classes, starting from the second incremental step, random oversampling is applied to the 2000 exemplars, bringing them to a total of $K' = 5000$. After having oversampled exemplars, the total amount of images with which we train

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	0.460	0.596
w/ AutoAugment	0.921	0.826	0.782	0.732	0.702	0.662	0.623	0.586	0.551	0.515	0.690
w/ AutoAug.+TenCrop	0.926	0.836	0.784	0.744	0.709	0.676	0.635	0.598	0.562	**0.531**	**0.700**

Table 13. Test accuracies achieved by using AutoAugment at each incremental step (over the number of classes)

iCaRL during an incremental step is 10000; finally, they are all augmented with the AutoAugment policy of CIFAR-10.

Since we are augmenting the training set with so many different operations, we choose to train for more epochs than the default 70 of iCaRL. In fact, we argue that the huge variability in the augmented dataset can let us train the model for more epochs without incurring in overfitting. We chose to train for 150 epochs; the learning rate is divided by 5 after 105 and 135 epochs (7/10 and 9/10 of all epochs, similarly to our experiment with the standard iCaRL). All other hyperparameters were set as in section 2.5.

We evaluated the model at the end of each incremental learning step, with and without TenCrop at test-time. The evolution of the test accuracies for each method are reported in tab. 13.

In conclusion, finding effective augmentation policies with AutoAugment was successful in improving generalization on unseen test images, leading to better performances on CIFAR-100 with respect to the iCaRL baseline (+7.1% on the final incremental step test accuracy, +10.4% on the average incremental accuracy) thus also contributing in preventing catastrophic forgetting.

5.6. Random Erasing

5.6.1 Theoretical Description

In this section, we introduce another data augmentation technique called Random Erasing. When a model is excessively complex, such as having too many parameters compared to the number of training samples, it might be weaker in generalization. In bad cases, the CNN model may exhibit good performance on the training data but fail when predicting new data. To improve generalization ability, Zhong *et al*. [16] study the occlusion phenomena. Occlusion is a critical influencing factor on the generalization ability of CNNs. When some parts of the object are occluded, a strong classification model should be able to recognize its category from the overall object structure. Therefore, to address this problem and to improve the generalization ability of CNN, we introduce an image augmentation technique known as Random Erasing.

During the training phase, the Random Erasing technique is performed with a certain probability. For each image used in training the model, we apply the Random Erasing method with a certain probability p, otherwise the image is kept unchanged. Random Erasing randomly selects a rectangle region I_e in an image I, and erases its pixels with random values. Assume that the size (or area) of the image is $S = W \times H$. The area of the erasing rectangle region is randomly initialized to a value S_e, where the ratio S_e/S is in range $[s_l, s_h] \subseteq [0, 1]$, specified by the user. The aspect ratio H_e/W_e of the erasing rectangle region is randomly initialized to a value r_e in range $[r_l, r_s]$, again specified by the user. The erasing rectangle region I_e will have height $H_e = \sqrt{S_e \times r_e}$ and width $W_e = \sqrt{S_e/r_e}$. Then, we randomly initialize a point $P = (x_e, y_e)$ in I. If $x_e + W_e \leq W$ and $y_e + H_e \leq H$, we set the new the rectangle region, with size $I_e = (x_e, y_e, x_e + W_e, y_e + H_e)$ as the selected rectangle region. Otherwise, the above process is repeated until an appropriate I_e is selected. When the region to be erased is selected, each pixel in I_e is assigned a random value in [0,255], respectively.

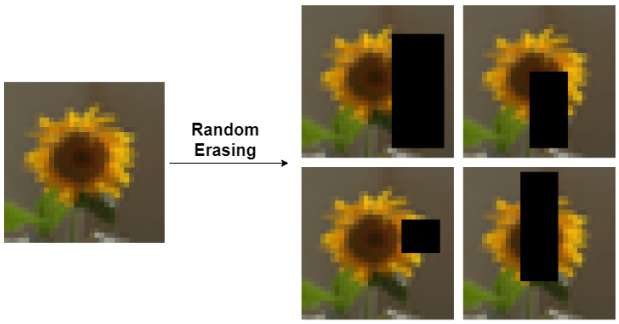

Figure 32. Random Erasing technique applied on an image of CIFAR-100

5.6.2 Experiments and results

Random Erasing data augmentation technique is used to virtually increase the cardinality of the dataset, in order to help the model to better generalize on new images. We apply this technique along with other basic transformations such as random crops and random horizontal flips, on all the training images belonging to the new classes and, from the second incremental step on, on all the exemplars. As for the hyperparameters of the Random Erasing method, we set $s_l = 0.02$, $s_h = 0.33$, $r_l = 0.3$, $r_h = 3.3$; moreover,

Method	10	20	30	40	50	60	70	80	90	100	Avg. incr. acc.
iCaRL.NME	0.868	0.738	0.664	0.617	0.577	0.553	0.523	0.487	0.476	0.460	0.596
w/ RandomErasing	0.863	0.765	0.703	0.656	0.629	0.594	0.55	0.513	0.506	0.482	0.626
w/ RandomEr. + TenCrop	0.884	0.784	0.715	0.658	0.631	0.6	0.56	0.525	0.51	**0.487**	**0.635**

Table 14. Test accuracies comparison

we fill the random erasing region with black pixels (value 0). We also repeated the same experiment, by using TenCrop at test-time. In order to address the imbalance between old and new classes, with random oversampling we increase the number of exemplars of the old classes from 2000 to 5000, prior to applying image augmentation operations. This technique outperforms iCaRL, both alone and by applying TenCrop. This means that with this tecnique iCaRL learns to extract features which are invariant to relatively small occlusions in the image, and this supposedly allows for a slower forgetting of the previous learned classes.

References

[1] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, 2018.

[2] Terrance DeVries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout, 2017.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[4] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[5] Saihui Hou, Xinyu Pan, Chen Change Loy, Zilei Wang, and Dahua Lin. Learning a unified classifier incrementally via rebalancing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[6] Hiroshi Inoue. Data augmentation by pairing samples for images classification, 2018.

[7] Guoliang Kang, Xuanyi Dong, Liang Zheng, and Yi Yang. Patchshuffle regularization. *CoRR*, 2017.

[8] Jiawen Kong, Thiago Rios, Wojtek Kowalczyk, Stefan Menzel, and Thomas Bäck. On the performance of oversampling techniques for class imbalance problems, 2020.

[9] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks, 2012.

[11] Zhizhong Li and Derek Hoiem. Learning without forgetting, 2017.

[12] L. O. Hall W. P. Kegelmeyer N. V. Chawla, K. W. Bowyer. Smote: Synthetic minority over-sampling technique, 2011.

[13] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, and Christoph H. Lampert. icarl: Incremental classifier and representation learning, 2016.

[14] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning, 2019.

[15] Lijuan Wang Yuancheng Ye Zicheng Liu Yandong Guo Yun Fu Yue Wu, Yinpeng Chen. Large scale incremental learning, 2019.

[16] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation, 2017.