Incremental Learning in Image Classification Report

Cordaro Nicolò S272145 | Di Nepi Marco S277959 | Falletta Alberto S277971

Abstract

In this report we describe our experience with some strategies of incremental learning, machine learning approaches tackling the current too limited ability of neural networks of learning new concepts over time.

The aim of incremental learning, in fact, is for the learning model to adapt to new data without forgetting its previously acquired knowledge. This is particularly useful in cases where, training a model from scratch every time new data are available, is too expensive both in term of money and time, or in cases where past training data become unavailable after a certain amount of time.

In this project we go through our implementation of two methods based on knowledge distillation (Learning Without Forgetting and ICaRL), some experiments on different loss functions and classifiers and possible suggestions based on our findings, trying to improve the model.

To do so we use CIFAR100, a dataset of 60000 32x32 colour images divided in 100 classes, with 600 images per class.

Introduction

Incremental Learning is an open problem in the field of machine learning and image recognition. Nowadays we are approaching the challenge of artificial intelligence and, as a consequence, we need to address a more dynamical context and interactive scenarios in which the dataset is not fixed and known apriori but more and more data and new classes are available over time.

Our goal is to find a robust model able to learn the new classes but without suffering from catastrophic forgetting and maintaining a limited number of past examples.

1. Catastrophic Forgetting Baseline [1]

This first experiment, starting point of the project, consists in the implementation of the finetuning baseline. We do not use any strategy to prevent the expected accuracy drop, called catastrophic forgetting, that happens when a model, trained multiple times on different batches of classes, is no longer able to correctly classify the first ones.

The training procedure is common to all the experiments and consists of training the model on 10 random classes of CIFAR each time, keeping track of the accuracy of the model on all the known classes throughout the process until all 100 classes have been used, that means 10 training phases.

1.1. Dataset creation

Before we begin, a common note for all the experiments: starting from TorchVision's CIFAR100 we decided to adopt a slightly different and more flexible implementation. We designed a class called CIFAR100_tError which can be seen as an empty container[2] that can be easily filled of images of given classes by mean of a function called *increment*. This function accepts as arguments the list of desired classes whose images are to be added to the dataset, along with a list of new indices to remap the original class labels since the model needs the distinct classes to be an ordered list of integers starting from 0.

1.2. Methodology

Parameters: We decided to show catastrophic forgetting using the same parameters implemented by Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, Christoph H. Lampert in iCaRL paper [2], that means each training step consists of 70 epochs, learning rate starts at 2.0 and is divided by 5 after 49 and 63 epochs, batch size 128 and a weight decay parameter of 0.00001.

Transformations: To be consistent with the original implementation we considered iCaRL's transformations namely, RandomHorizontalFlip, Pad(4), RandomCrop(32) and the normalization using ImageNet's mean and standard deviation instead of CIFAR100's since we noticed a slight accuracy improvement.

[1] Code available at `https://github.com/Nicordaro/Project_MLDL`

[2] The object actually has all the images loaded in a variable called "dataset". When increment function is called the images of the desired classes are then copied in "data" which is the actual dataset variable

Network: Although iCaRL made use of a ResNet32 We have chosen instead a smaller and slightly faster ResNet18 which surprisingly provided us with even better results. The network is a modified version of Pytorch's one adapted to receive as input 32x32 images.

Functions: In the provided code we use three functions, namely, "make_data_labels", "testNet", "accuracy_plot". The first is used to randomly select the 10 classes to train the model on, in fact accepts a list of integers already in pseudo-random order corresponding to all the possible classes and returns two lists, one containing all the elements but the first ten, the other containing the first ten. The second function is simply used to test the network and the third to plot results.

Training: In order to show catastrophic forgetting of the network we use a *for* cycle to train the model on 10 new random classes each time. Every iteration 10 new random classes are selected by mean of "make_data_labels" function and then loaded into the training dataset, into the evaluation dataset (which is actually a test dataset with only images belonging to the 10 training classes) and into the test dataset (which contains test images of the current 10 training classes and all the images belonging to the classes already used for training). It is a trivial training and testing procedure apart from the fact that for every epoch the loss is computed on a one hot encoding of the labels since the loss used is BCEwithLogitsLoss. Furthermore we decided to test the network after every epoch on the images of the evaluation dataset in order to keep track of the best performing model to use it for the accuracy computation on the test dataset, this is not a completely fair validation since it is performed on the evaluation dataset that is actually made of test images, but is a practice accepted in this field.

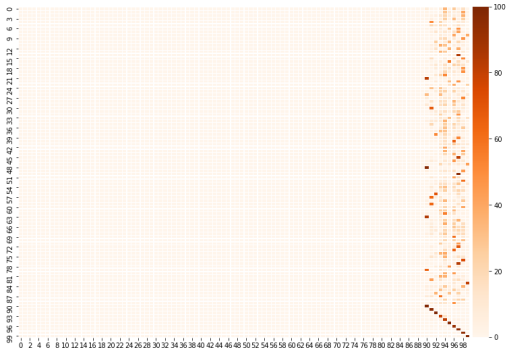

Figure 1. *Confusion Matrix for finetuning*

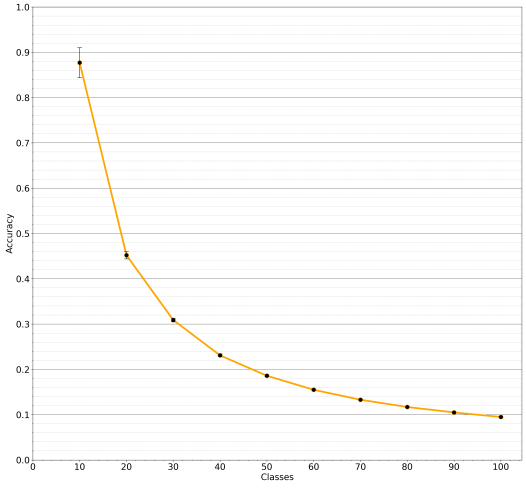

Figure 2. *Accuracy plot for finetuning*

1.3. Results

Figure 2 provides the confusion matrix of the final test accuracy with all the 100 classes. It shows that the finetuned network at the end of the process only recognizes classes of the last training batch and completely forgets the existence of the others. As expected, the accuracy drops dramatically below 10% as showed in figure 1.

2. Incremental Learning Strategies

In this section we describe two strategies for the incremental learning: Learning Without Forgetting and iCaRL. Both are implemented following the details in iCaRL paper.

2.1. Learning Without Forgetting

This first strategy was proposed by Li and Hoiem [2] for a multiple tasks problem and is adapted here in the field of multi-class learner. It adopts the principle of distillation: to address catastrophic forgetting the network is stimulated with both new and old data trying to avoid the deterioration of the information. In practise, the training strategy remains unchanged but the loss is now made of two terms: the classification loss between the network's output and the new encoded labels and the distillation loss between the output of the network for the old classes and the output of the old network, previously saved before the training. Both terms are implemented by using a binary cross entropy loss function. Learning Without Forgetting improves the performances (figures 3 and 4) and the network is able to maintain memory of past classes a little longer, however, we are still far from acceptable results since the accuracy after the last batch is about 20% and the bias towards the new classes is still very much present.

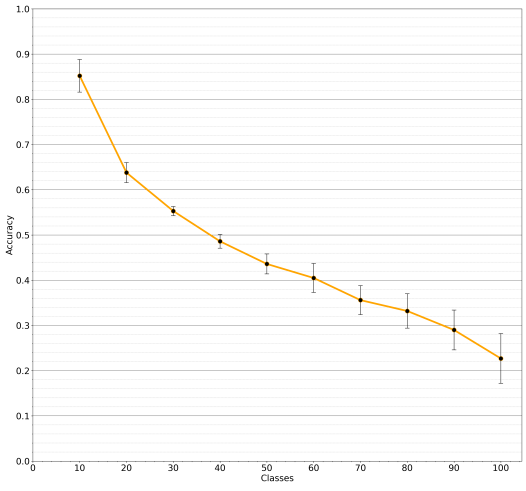

Figure 3. *Mean and standard deviation of the accuracy values obtained with LwF*

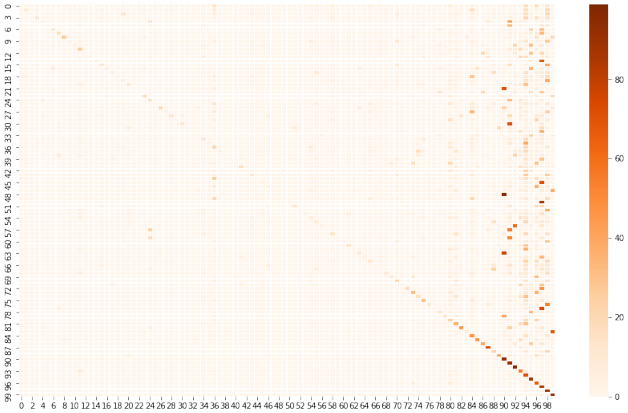

Figure 4. *Confusion matrix LwF*

2.2. iCaRL

A different strategy is proposed in iCaRL. The novelties introduced in this paper, besides distillation loss, are classification by nearest-mean-of-exemplars (NME) and the use of exemplars. NME classifier, used in the classify function, consists in computing the prototype vector for each observed class (the mean of the extracted features) and assigning to each image the label of the most similar vector, making the classifier more robust with respect to the the usual fully connected layer.

The reason behind this choice is that we would like the weights of the last layer to change simultaneously with the parameters of the feature extractor. Usually, this is not possible without data from previous classes (the authors named this issues as decoupling of φ and w). Instead, Nearest Class Mean solve the problem by computing the means whenever a training phase is completed so that even classes from different batches are classified correctly.

On the other hand, exemplars ensure to have at least a few previous classes images and are organized in a prioritized list following the principle of herding to choose the best possible representative subset: since the resources are limited the number of exemplars that can be memorized is bounded to a memory constraint K, such that whenever the network encounters new classes, the exemplars sets already collected are reduced and the last elements of each list (less representative ones) are discarded.

2.3. Methodology

We started from the previous ResNet18 that for simplicity is divided in two parts: A features extractor (convolutional layers) and a classifier (fully connected layer). The five functions are implemented as detailed in iCaRL paper.

- **Classify**: Given a batch of images finds the nearest prototype and returns the labels. In our case, instead of computing the means for only the exemplars, we

used all the images available in that moment.

$$y* \leftarrow \underset{y=1..t}{argmin} \|\varphi(\mathbf{x}) - \mu_{\mathbf{y}}\|$$

- **Incremental Train**: Main function that calls all the others and prepares the datasets needed.

- **Update Representation**: The training is implemented in this function along with the computations of losses and creation of the training set, that is the combination of the new data and the exemplars.

$$\ell(\phi) = -\sum_{x_i, y_i \in D} \sum_{y=s}^{t} \delta_{y=y_i} \log g_y(x_i) + \delta_{y \neq y_i} \log(1 - g_y(x_i))$$

$$+ \sum_{y=1}^{s-1} q_i^y \log g_y(x_i) + (1 - q_i^y) \log(1 - g_y(x_i))$$

- **Construct Exemplar Set**: For each new class, build a list of most representative images making sure that each image appears at most once.

  for k = 1,...,m **do**

$$p_k \leftarrow \underset{x \in X}{argmin} \|\mu - \frac{1}{\mathbf{k}} [\varphi(\mathbf{x}) + \sum_{\mathbf{j=1}}^{\mathbf{k}} \varphi(\mathbf{p_j})]\|$$

  end for

- **Reduce Exemplar Set**: For each exemplar sets, keep only the firsts m elements and remove the others.

The hyperparameters used are the same described in section 2.2 since they are the best accuracy achieving ones while the memory size K is set to 2000.

2.4. Results

With iCaRL there is a clear accuracy improvement, figure 5 and 6 show how this varies with the number of classes and the confusion matrix respectively. This shows that iCaRL is able to classify in a more homogeneous way without suffering from catastrophic forgetting. In conclusion this is a good strategy but the results are still far from the ones that could be achieved by training the model on all the classes at once, furthermore it is highly dependant on the use of the exemplars, that is a debated topic in the field.

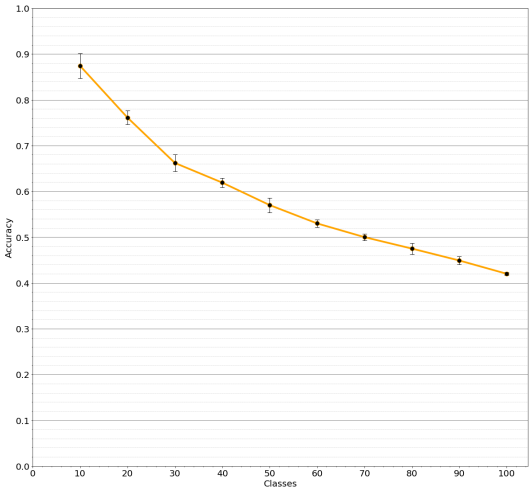

Figure 5. *Mean and standard deviation of the accuracy values obtained with iCaRL*

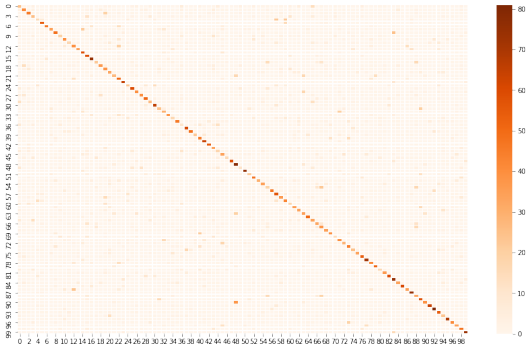

Figure 6. *Confusion matrix shows an homogeneous classification for all the classes*

3. Ablation Study on Losses

In this section we will make some experiments with different combinations of classification and distillation losses, trying to find a better alternative to the binary cross entropy. L1, L2 and Cosine embedding are multiplied by a scalar factor which serves to give greater weight to the loss and increase the distillation effect. For these last 3 cases The learning rate is set to 0.02 and we used the Cross Entropy Loss as classification loss because it's generally bigger than the BCE and easier to compare with the others.

3.1. Kullback-Leibler Distance

Kullback-Leibler divergence is used to measure the distance between two probability distributions. Input and Target are then the softmax of the outputs of the new model and the ones of the old model. We expect this loss to be good since minimizing Kullback-Leibler is the same of minimizing the Cross Entropy. For the classification loss instead, we kept the BCE.

$$l_n = y_n \cdot (\log y_n - x_n)$$

3.2. L2 Loss

The mean squared error is an interesting choice. It is commonly used for regression problems but in this case we applied it between the normalized features extracted by the convolutional layers of the two models. This is reasonable since we use the features to do the classification with NEM. Due to the square we are penalizing more big errors while small differences bring to a very small loss.

$$l_n = (y_n - x_n)^2$$

3.3. L1 Loss

L1 loss measures the mean absolute errors between the input and the target. We made two distinct experiments: in the first, as before, we computed the loss between the extracted features, in the second the error is computed between the outputs of the fully connected layers. This is a simpler model and rarely used, so we expect it to give worse results.

$$l_n = |y_n - x_n|$$

3.4. Cosine embedding Loss

The goal of this loss is to maximize the cosine similarity between two vectors. The idea is proposed by Hou et al. in the paper *Learning a Unified Classifier Incrementally via Rebalancing*[4] with the name of less forget constraint. The reasoning behind is to preserve the spatial configuration of the previous knowledge and encourage the new features to be similar to the old ones. The weight of the loss is not fixed, but varies with the number of known classes. Since the features are also normalized this loss considers only the orientations and the angles between the vectors and not their magnitude making the model more flexible.

$$l(x_1, x_2) = 1 - cos(x_1, x_2)$$

3.5. Results

The table reports accuracy values obtained with the five combinations of losses. As expected, with Kullback Leibler divergence we obtain a curve similar to the standard iCaRL, with a few percentage points less on the last batches that may be only due to the hyperparameters.

The L1 loss between the logits instead leads to a worsening of about 10 percent on the last batch. This is reasonable because L1 loss does not punish enough misclassifications and is more suitable for regression problems and continuous targets.

Regarding the loss computed between the features, L1 performs better than both L2 and the cosine embedding loss. A possible explanation to the first case is that L2 due to the presence of the square is more sensitive to outliers and may result in big deviations. On the other hand, minimizing cosine similarity is not so effective in a scenario in which the other components of the model (Cosine normalization layer and Margin ranking loss) in paper [4] are not implemented.

Overall, it can be noticed that despite the huge drops in the first batches of classes, the degradation of the information is limited on the long run.

Table 1. *Mean accuracy values for each step*

	KLD	L1(FC)	L1(Feat)	L2	COS
20	0.74	0.68	0.65	0.60	0.61
30	0.66	0.58	0.54	0.54	0.50
40	0.59	0.49	0.49	0.47	0.41
50	0.55	0.44	0.46	0.42	0.37
60	0.50	0.39	0.43	0.40	0.34
70	0.47	0.36	0.42	0.38	0.32
80	0.45	0.35	0.39	0.36	0.31
90	0.41	0.33	0.37	0.33	0.30
100	0.37	0.30	0.34	0.30	0.27

4. Ablation Study on Classifiers

In this section we will study possible alternative to the nearest class mean. In order to keep things comparable, we decided to change the classifier without varying the losses and the structure of the training of ICaRL.

4.1. Bias Correction Layer

The solution comes from Wu et al [5] in the paper *Large Scale Incremental Learning*. The main problem of the fully connected layer is that it is biased towards the new classes due to the different balance in number of data between new classes and exemplars. The training is then divided in two parts: In the first stage the network is trained up to the fully connected layer, in the second stage it is used a small balanced validation set (in our case implemented using train test split with stratify from Sklearn) to learn two new parameters, alpha and beta. These values are used to estimate the modified output for the new classes with a simple linear model as showed in the formula below.

$$q_k = \begin{cases} o_k, & \text{if } 1 < k < n. \\ \alpha o_k + \beta, & \text{otherwise.} \end{cases} \quad (1)$$

where o_k is the logit output for class k. In the second stage we used a small learning rate (0.001) with a different optimizer and 300 samples for the validation set.

4.2. Cosine Layer

Another interesting proposal presented in [4] to solve the problem of imbalance is cosine normalization. Typically, the last layer of the network have weights and biases that

are significantly higher for the new classes compared to the old ones. Then, they decided to normalize both the features extracted and the weights and compute the cosine similarity between them. Since the result is limited in [-1,1] they also introduced a learnable parameter η to control its peakiness.

Therefore, the output of the last layer changes from:

$$\theta_i^T f(x) + b_i$$

to:

$$\eta < \bar{\theta_i}^T, \bar{f}(x) >$$

In the paper the authors introduced, in addition to the cosine layer two other components (that we did not implement in this part):

1. Less forget constraint, as we talked before, is an alternative distillation loss that aims to preserve the orientation of the features
2. Margin ranking loss to ensure that the old features cannot be confused with the new features

4.3. Naive Bayes Classifier

Since Sklearn provides classifiers able to learn incrementally we decided to experiment replacing the Nearest-Class-Mean classification with a classifier with the aforementioned property. The family of Naive Bayes classifiers are simple probabilistic classifiers based on Bayes theorem with strong and naive independence assumptions. Given an item with set of features X the we are able to compute the probability of the item belonging to a class c using the Bayes theorem:

$$p\left(c|X\right) = \frac{p\left(X|c\right)p\left(c\right)}{p\left(X\right)}$$

Where $p\left(c|X\right)$ is the posterior probability, $p\left(X|c\right)$ the likelihood, $p\left(c\right)$ the prior probability of class and $p\left(X\right)$ the prior probability of predictor. Unfortunately the likelihood can be hard and expensive to compute. Therefore, the previous formula can be simplified both from the naive assumption (explanatory variables X are assumed to be independent from the target variable c) and its consequences on the calculation of the likelihood and also from the prior probability of predictor $p\left(X\right)$ which is constant given the input so that the posterior probability can be written as:

$$p\left(c|X\right) = p\left(x_1|c\right)p\left(x_2|c\right)...p\left(x_n|c\right)p\left(c\right)$$

Every time a group of images belonging to 10 new classes is given, the network trains all its layers first (convolutional layers and fully connected layer) and then it trains the Naive Bayes classifier using the normalized features representations of the same training images given by the feature extraction layers just trained at the previous step.

4.4. Results

Figure 7. shows the trends of the different tested methods. Despite nearest mean still being the best classifier so far, the results achieved with the variations proposed are not too far. Cosine Normalization and Bias correction, that are in practise an improved version of a fully connected layer, perform similar and follow almost the same curve, with the first being slightly better in the first five batches and the second taking the upper hand the in the last five. Notwithstanding the premises, these methods without their corresponding loss are not able to fully address the problem of a too biased classification.

The incremental classifier proposed, Naive Bayes classifier, have a more steep curve in the first part of the process, but after the sixth batch is able to outperform the other two.

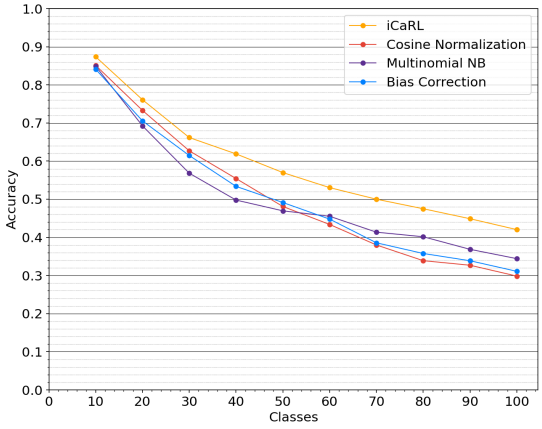

Figure 7. *Comparison of the accuracy trend between Icarl (nme) and the variations*

5. Variation

One of the main weak points of iCaRL is the unbalanced dataset used for the training of the network. Indeed, during the last phase there are only 22 images for the old classes compared to the 500 of the new ones and this is one of the causes of the accuracy drop. The hypothesis is confirmed since loosening the memory constraint, from 2000 to 4000 exemplars, shows an improvement in the results. So, we decided to focus on the augmentation of the exemplars at training time without storing them in the memory.

5.1. Related works

Our method is mainly inspired by advanced augmentations techniques on Cifar dataset and some other papers that tried to improve iCaRL.

Ekin D. Cubuk et al. [6] propose **AutoAugment** to auto-

matically search for the best augmentations policies. Their work is based on Reinforcement Learning and uses a Recurrent Neural Network as a controller for a small child network. The result, in the Cifar case, is a policy made of 24 sub-policies. Each sub-policy is a set of two transformations that come together with a probability value (the probability that the transformations will be applied) and a magnitude value.

Inoue's **Data Augmentation by Pairing Samples for Images Classification**[7] is another interesting paper about augmentation. The strategy behind is simple but effective: Given a small dataset, new images can be generated overlaying two random images and taking the average of the values at pixel level. For the label of the resulting image is used the one of the first image taken.

Finally, in **End-to-End Incremental Learning** [8] Castro et al. implemented, in addition to new transformations compared to those used in iCaRL, a balanced fine-tuning phase using a small subset of the most representative among the new images along with the most representative from the old classes.

5.2. Methodology

In addition to HorizontalFlip and RandomCrop we added a CIFAR100Policy to the exemplar images. When the function is called, it randomly chooses one sub-policy and applies it to the mini batch.

After the second batch of classes, we created also at training time a new augmented dataset that contains also the new exemplars generated by the sample pairing function. These new samples are not saved but are removed after the last epoch in order to remain under the memory size constraint. However, the resulting size of the exemplars set virtually grows to have at least 400 images per class.

Starting from the stardard iCaRL, we divided the training in three parts. The Learning rate and the other hyperparameters are unchanged.

1. Training for 30 epochs without Sample Pairing
2. Training for 40 epochs with Sample Pairing active intermittently disabling it after 8 epochs and for the next 2 epochs
3. A final fine-tuning on the exemplar sets for 15 epochs and a very small learning rate, like 0.0001.

Despite our efforts, this method do not provide any significant accuracy improvement on the test set, but only a slightly decrease after the fine-tuning.

A possible explanation may be that the images generated do not introduce enough variation in the data and since the classification is still made taking the mean of the real exemplars this modification does not affect the results too much.

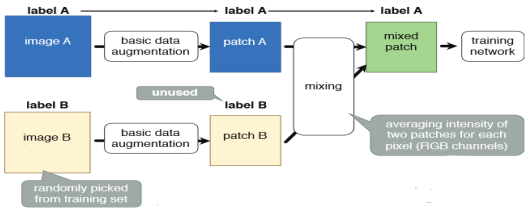

Figure 8. *Sample Paring images generation*

6. Conclusion

In this project we studied the problem of incremental learning and how the various issues are addressed in literature. Starting from the finetuning baseline, in which the network was unable to remember the past classes, we implemented Learning Without Forgetting and iCaRL. The latter used exemplars and nearest class mean to avoid catastrophic forgetting but we went deeper making experiments with different combinations of losses and classifiers that did not bring better results. Finally, we made an attempt to generate new images based on data augmentation strategies.

Notwithstanding the improvements in the last years, these methods don't reach the performances of a joint-training and massive effort of research is still ongoing.

References

1. Geoffrey Hinton, Oriol Vinyals, Jeff Dean. Distilling the Knowledge in a Neural Network. In NIPS Deep Learning Workshop, 2014.
2. Z. Li and D. Hoiem. Learning without forgetting. In European Conference on Computer Vision (ECCV), 2016.
3. Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl and Christoph H. Lampert. iCaRL: Incremental Classifier and Representation Learning. In Computer Vision and Pattern Recognition (CVPR), 2017.
4. Saihui Hou1, Xinyu Pan, Chen Change Loy, Zilei Wang, Dahua Lin. Learning a Unified Classifier Incrementally via Rebalancing. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019
5. Yue Wu, Yinpeng Chen, Lijuan Wang, Yuancheng Ye, Zicheng Liu, Yandong Guo, Yun Fu. Large Scale Incremental Learning. In Computer Vision and Pattern Recognition (CVPR), 2019.
6. Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, Quoc V. Le. AutoAugment: Learning

Augmentation Policies from Data. In Computer Vision and Pattern Recognition (CVPR), 2019.

7. H. Inoue. Data augmentation by pairing samples for images classification. 2018

8. Francisco M. Castro, Manuel J. Marín-Jiménez, Nicolás Guil, Cordelia Schmid, Karteek Alahari. End-to-End Incremental Learning. In Computer Vision and Pattern Recognition (CVPR), 2018.

7. policies found on CIFAR with AutoAugment

Table 2. *Sub-policies taken from the AutoAugment paper*

	Operation 1	Operation 2
Sub-policy 0	(Posterize,0.4,8)	(Rotate,0.6,9)
Sub-policy 1	(Solarize,0.6,5)	(AutoContrast,0.6,5)
Sub-policy 2	(Equalize,0.8,8)	(Equalize,0.6,3)
Sub-policy 3	(Posterize,0.6,7)	(Posterize,0.6,6)
Sub-policy 4	(Equalize,0.4,7)	(Solarize,0.2,4)
Sub-policy 5	(Equalize,0.4,4)	(Rotate,0.8,8)
Sub-policy 6	(Solarize,0.6,3)	(Equalize,0.6,7)
Sub-policy 7	(Posterize,0.8,5)	(Equalize,1.0,2)
Sub-policy 8	(Rotate,0.2,3)	(Solarize,0.6,8)
Sub-policy 9	(Equalize,0.6,8)	(Posterize,0.4,6)
Sub-policy 10	(Rotate,0.8,8)	(Color,0.4,0)
Sub-policy 11	(Rotate,0.4,9)	(Equalize,0.6,2)
Sub-policy 12	(Equalize,0.0,7)	(Equalize,0.8,8)
Sub-policy 13	(Invert,0.6,4)	(Equalize,1.0,8)
Sub-policy 14	(Color,0.6,4)	(Contrast,1.0,8)
Sub-policy 15	(Rotate,0.8,8)	(Color,1.0,2)
Sub-policy 16	(Color,0.8,8)	(Solarize,0.8,7)
Sub-policy 17	(Sharpness,0.4,7)	(Invert,0.6,8)
Sub-policy 18	(ShearX,0.6,5)	(Equalize,1.0,9)
Sub-policy 19	(Color,0.4,0)	(Equalize,0.6,3)
Sub-policy 20	(Equalize,0.4,7)	(Solarize,0.2,4)
Sub-policy 21	(Solarize,0.6,5)	(AutoContrast,0.6,5)
Sub-policy 22	(Invert,0.6,4)	(Equalize,1.0,8)
Sub-policy 23	(Color,0.6,4)	(Contrast,1.0,8)
Sub-policy 24	(Equalize,0.8,8)	(Equalize,0.6,3)