Incremental Learning Project Report

Eleonora Carletti
ID: s280056
s280056@studenti.polito.it

Lucia Innocenti
ID: s279123
s297123@studenti.polito.it

Abstract

The challenge of the Artificial Intelligence is strictly related to the problem of incremental learning. In this paper, we have tried to reproduce LWF and ICaRL, two important cornerstones in studies to avoid catastrophic forgetting. Then we have proposed some variations to ICaRL in order to better analyze its weaknesses and highlight some possibilities of future improvements. The implementation is available in this GitHub repository[1]

1. Introduction

The main goal of this project is to implement a neural network capable of learning different tasks without forgetting the ones it has learned before. This is a very important goal for neural network, the one that most approaches neural networks to artificial intelligence. It is a relatively new problem, so in literature there are not so many papers. We can formalize the purpose of this work in the following way: given a CNN with certain shared parameters θ_s, add task specific parameters θ_n for a new task that works well both on old and new tasks. There are some constrains that defines that the network is incremental learning new tasks:

- The model has to adapt gradually i.e. is constructed based on without complete retraining.
- Preservation of previously acquired knowledge and without the effect of catastrophic forgetting.
- Data from previous tasks are not available when new tasks are processed.

We will try to reproduce two bases approaches:

- Learning without forgetting,
- ICaRL

[1] https://github.com/luciainnocenti/IncrementalLearning

2. Tools and libraries used

In this section will be described the libraries and the tools used during the execution of the project. In particular the tools analyzed will be Github, Google Colaboratory, Pytorch library and will be presented an overview of the most famous machine learning libraries.

2.1. Github

GitHub is a code hosting platform for collaboration and version control. It is based on the version control system Git and it lets people create cloud hosted repositories and work together on them. In the project the platform has been used to host repositories and files. The code is available at this link: https://github.com/luciainnocenti/IncrementalLearning

2.2. Google Colaboratory

Google Colaboratory (or Colab) is a free cloud service providing machines running several GPU's and CPU With this platform it is possible to develop deep learning applications using popular libraries such as Keras, TensorFlow, PyTorch, and OpenCV. In the project the platform has been used to host the train and test logic of the network implemented since this kind of operations requires huge amount of computational GPU power.

2.3. Pytorch

PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language. In the project the library's functions has been used mainly to instantiate the networks and performing train, evaluation and test phases. Other functions of the library have been used to perform minor importance operations.

2.4. Commonly used python libraries

In this project have been also used other libraries commonly found in several machine learning projects.
Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and ma-

trices, along with a large collection of high-level mathematical functions to operate on these arrays.
MatPlotLib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits In this project will be used to plot graphs.

3. CIFAR 100 Dataset

3.1. The Dataset

The CIFAR 100 dataset is an extension of the most famous CIFAR 10 dataset. It has developed by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton and consists of 100 classes containing 600 images each. The classes are grouped into 20 super-classes and there are 500 training images and 100 testing images per class. Each image comes with a "fine label" (the class to which it belongs) and a "coarse" label (the super-class to which it belongs). It also contains the `label.names` file that maps each class to a human readable name. The main features of the dataset are summed up in *Figure 1*. In this project only the classes and the fine labels will be considered. In particular groups of ten classes each time will be used to feed the project's networks.

3.2. Implementation Details

In the implementation of the dataset the object `self_dataset` is defined. The object belongs to the class `Cifar100` which extends the `Cifar10` class. It contains the following elements: Data, Targets, Classes, Class_to_index.
Data elements refers to all the images contained by the dataset. **Targets** elements are the unique numbers identifying the classes. **Classes** elements are the classes names. **Class_to_index** is a dictionary containing the index as a key and the class as a value.

The dataset also provide some standard methods, like **__len__** and **__getitem__** and an additional method, **returnSplits**, that, given the indices of all the classes, split them into 10 random groups. The model will not store also the images of the related split, but all the operations are executed by managing indices. CIFAR100 contains an heterogeneous set of classes; the random split is a very important operation and it very affected the performances for all our works. So, because of the high variability by changing splits, we have selected one that gave good results in fine-tuning and we have used it for all the methods implemented; by using this approach, we are sure to have comparable results from one implementation to another.

The base structure used in order to manage the dataset is:

- Create the dataset containing the 100 classes;

Total Images	Classes	Superclasses	Images per class	Train images per class	Test images per class	Size
60000	100	20	600	500	100	32x32

Figure 1. Cifar 100 Dataset features

- By using the classes associated with the current task, defined by the method *returnSplits*, retrieve all the indices of the images belonging to those classes;

- Use the method subset to obtain a partial dataset, composed by the images identified so far;

- Create the correspondent data-loader and train the network;

This approach, however, presents a problem when using data augmentation. The "subset" method defined in Pytorch is created to select, from the original dataset, all the images defined in the index array. But in this way, data augmentation is applied to the whole data set and not to the partial one. Therefore, the effects of data augmentation are reduced.

To solve this problem, a new implementation of the "subset" method has been defined, which applies the transformation method directly to the elements returned in that specific subset call.

4. Fine Tuning

In order to highlight the problem of catastrophic forgetting in neural networks, we have tried to implement the incremental learning without using any kind of specific attention. So, the model is a ResNet as described above; the dataset is the CIFAR100. As loss function a BCE was selected and the classifier is a simpler linear one: the class with the higher output probability is selected.

4.1. Implementation Details

The structure, that is used also for other implementations, is the following: for 10 times, groups of 10 classes are passed through the network to train it. These group are concatenated in the test phase. So, in the training phase, the network see ten classes and it is trained by learning their features; in the test phase the network see all the classes analyzed so far (i.e. $10 \cdot currentTask$ classes).

Being the goal of the project to compare results among different implementations, some elements are in common with all the approaches used:

- Optimizer: SGD (Stochastic Gradient Descent), it optimizes all the parameters in the resNet, as hyper-parameters take the learning rate, the step size and the weight decay.

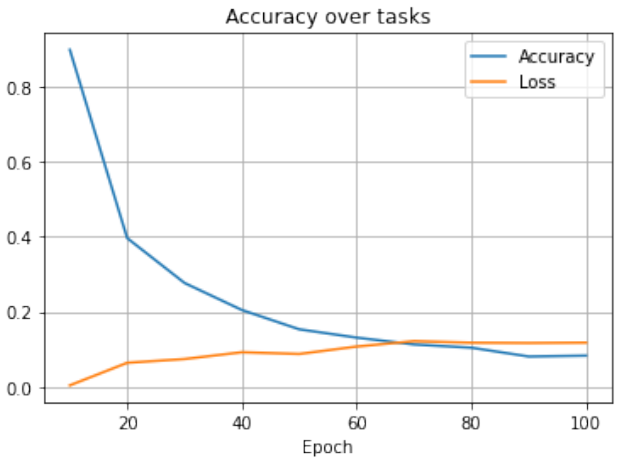

Figure 2. Accuracy and Loss - Finetuning

- Scheduler: the MultiStepLR defined in Pytorch changes the learning rate by multiplying it to a constant γ as some fixed epochs

4.2. Results

The model trained without any specific implementation highlights the effect of catastrophic forgetting. As it's possible to see in *Figure 2*, the accuracy obtained by training and testing the model on the same group of classes in very good (around 90%). But, at the second step, it collapses at 40%, less than the middle of the first one. And the trend continue along the steps. The final task, when the network is trained on the last 10 classes and tested on the whole dataset, the accuracy is around the 10%.

5. LWF model

The model [3], first proposed by Zhizhong Li and Derek Hoiem in 2017, is an attempt of solution of one of the most common problems in multi-class learning: catastrophic forgetting. It exploits the idea of distillation loss, that was originally used to reduce the differences between two networks. In LWF the distillation loss is used to lower the differences between the same network at two different steps.

5.1. Theory Overview

In common convolutional neural networks there exist three types of parameters. The first ones are the shared parameters θ_s that corresponds to the convolutional layers. The second type of parameters θ_0 are the task specific parameters for previously learned tasks. The last ones , θ_n, are randomly initialized task specific parameters for new tasks. The network structure is represented in *Figure 3*.

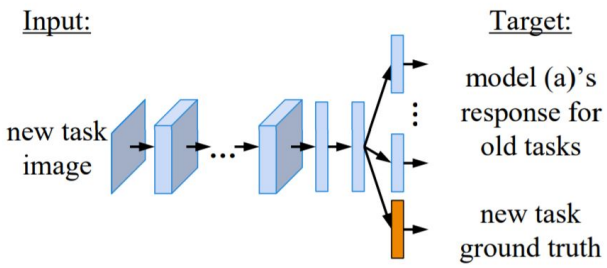

Figure 3. LWF Network structure

5.2. Implementation Details

The distillation loss used in this project is a special version created for ICaRL that, instead of sum a classification loss (calculate by new elements and their labels) and a distillation loss (calculate on the outputs of the freezed net and the unfreezed one on the new images) , concatenate these two parts and calculate the loss in a single step. So, the obtained result is that the output of the updated network (which has dimension $BATCH\_SIZE \cdot NUM\_CLASSES$, has "splitted" in three parts:

- columns referred to old classes, for which the loss is calculated by comparing them to the same columns from the output of the freezed net;
- columns referred to classes of the current task, for which the loss is calculated by comparing the to the One-Hot version of the labels
- columns referred to future classes, for which the value is zero for both output and one-hot, so these columns not infer the loss.

In order to collect the data from the network at the previous step, after each training phase the network are stored and, at the beginning of every train and test phase, the last saved net is downloaded twice: one will be updated and will became the freezed network of the successive step.

5.3. Results

The LWF model performances are reported in *Figure 4* It's possible to see that there is a big improvement with respect to the fine-tuned model. Especially from the differences between the first and the second batch, where it can be seen that the accuracy is not halved from one to the other. But the final result is anyway very low: around the 25% of accuracy. So we can get out that the distillation loss is a very good improvement, but it's not enough to solve the catastrophic forgetting problem.

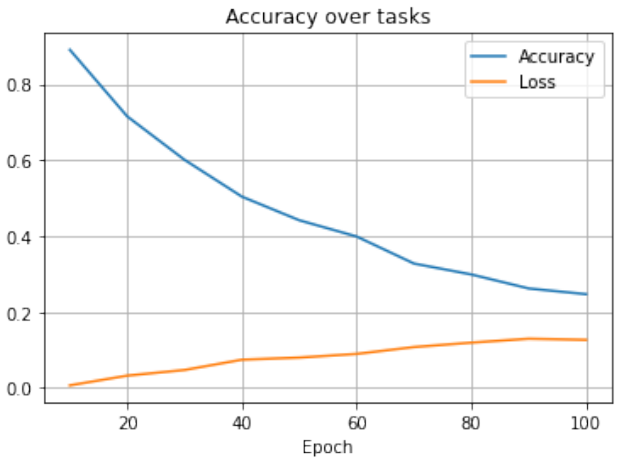

Figure 4. Accuracy and Loss - LWF

6. I-Carl Model

The model [1], proposed by Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl and Christoph H. Lampert in 2017, is another attempt of solution of the catastrophic forgetting problem. As LWF it exploits the concept of distillation loss, but differently to it, feeds the network not only with images and labels belonging to the new task but also with selected images from the previously learned tasks (the "exemplars"). Thanks to this combination of new images and old images the network is able to better tune its internal hyper parameters to avoid forgetting old tasks.

6.1. Theory Overview

In old machine learning projects, whenever a problems was too hard to be computed all in a row the common approach was to split the problem in several sub tasks and find different strategies to solve them independently.
But this was an inefficient strategy since it was subject to the catastrophic forgetting issue. What I-Carl is aiming to do is to train a sort of multiclass learner that is able to take data from all the tasks and perform better predictions by making use of shared information.
Before I-Carl, multitask learning strategies had the problem of having access to all tasks data at the same time during training (*Figure 5*) and this didn't let the networks to learn new things in an incremental way (*Figure 6*).
ICarl provides an implementation of the multi-class learner that is capable to learn new task incrementally. To do so the network take as an input a stream of data in which examples of different classes occur at different times and produces as an output a multi-class classifier for all the classes observed so far. All this operations are performed under the restraint that computational requirements and memory footprint remains bounded (or at least grows very slowly) with respect to the number of classes seen so far. The network is com-

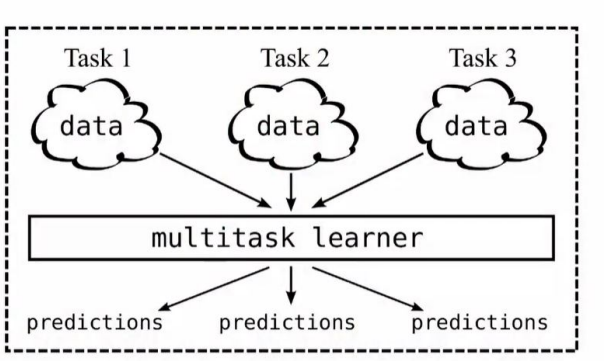

Figure 5. Traditional Multiclass Learner

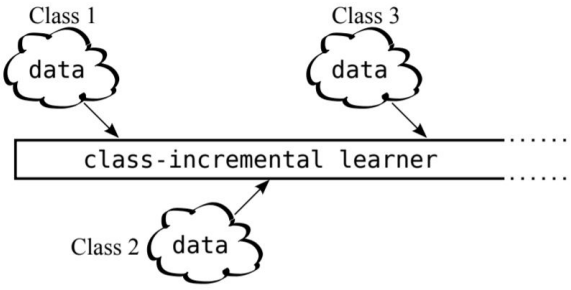

Figure 6. Incremental Multiclass Learner

posed by the following three components:

- Classification by a nearest mean of exemplars rule
- Prioritized exemplar selection based on herding
- Representation learning using knowledge distillation and prototype rehearsal.

For classification, iCaRL relies on sets $P_1, ..., P_t$, of exemplar images that it selects dynamically out of the data stream. The total number of samples should never exceed a fixed number denoted as **K**. As a classification strategy iCarl uses the **Nearest Mean Of Exemplars**. To predict a label y^* for a new image x it computes a prototype vector for each class observed so far $\mu_1, ..., \mu_t$ where $\mu_y = \frac{1}{|P_y|} \sum_{p \in P_y} \varphi(p)$ is the average feature vector of all exemplars for a class y. It also computes the feature vector of the image that should be classified and assigns the class label with most similar prototype in this way:

$$y^* = \underset{y=1...t}{\operatorname{argmin}} ||\varphi(x) - \mu_y||$$

For what concerning learning iCaRL processes batches of classes at a time using an incremental learning strategy. Every time certain data $X_s, ..., X_t$ for new classes $s, ..., t$ is

available iCaRL calls an update routine that adjust its parameters and and exemplars according to the current training data. The same method, called "Representation Learning" is used to learn about the existence of new classes. The algorithm consists in constructing a combined training set consisting of the currently available training examples together with the stored exemplars. Then the current network is evaluated for each example and the resulting network outputs for all previous classes are stored. Notice that the output won't contain new classes since the network has not been trained for them yet. Finally, the network parameters are updated by minimizing a loss function that for each new image encourages the network to output the correct class label for new classes (classification loss), and for old classes, to reproduce the scores stored in the previous step (distillation loss). So it is possible to say the classification part is committed to learning new classes while the second one is committed to not forgetting the outputs of what has happened before.
Regarding the architecture iCarl is composed by a trainable feature extractor $\varphi : \chi \rightarrow \mathbb{R}^d$ followed by a single classification layer with as many sigmoid output nodes as classes observed so far. All feature vectors are L2-normalized, and the results of any operation on them are also re-normalized. It is possible to notice that iCaRL uses the network only for representation learning and not for the classification step. Finally regarding the resources usage topic the network doesn't need previous information about the number of classes that will occur during time. In particular the memory requirement will be equal to the size of the feature extraction parameters plus the storage of K exemplar images and as many weight vectors as observed classes.

6.2. Implementation Details

The main component of ICaRL implementation are the network, the train phase and the final classification phase.

6.2.1 The network

The network choose for the project is a ResNet32. The implementation of ResNet is modified in order to make available the possibility (mandatory for ICaRL) of manage both features and outputs. So the "forward operation" can be stopped at the level earlier of the last fully connected. This is done by using a flag that, if is set as True, allow the function to return the features; else it returns the outputs of the fully connected layer. The last FC layer is initialized at 100 outputs, the total number of classes available in CIFAR100.

6.2.2 Train Phase

ICaRL is based on the concept of exemplars. In order to avoid catastrophic forgetting, some elements from previous classes are stored and, during the training phase, these are concatenated to the dataset of new images and the network is trained on the dataset obtained. In order to have a more efficient algorithm, exemplars are created by storing the index of the image in the original dataset. Every time that a new class are met, the list of exemplars from previous classes are reduced in order to free space for elements of the new class. Three functions manage all these parts:

- **UpdateRep:** the function take as input the indexes of images of the new classes, the indexes of exemplars of previous classes, the network and return the network update by passing through the network all these images. It's done by using our implementation of the method "Subset" explained in the dataset section. So, given the new dataset, it should calculate the loss and backpropagate it. The loss function required to have both output from the network freezed at the earlier step and the output from the update network. So for each batch these values (outputs and oldOutputs) are calculated and given as input to the function *calculateLoss*, that is the same of LWF. The loss obtained as described are back-propagate among the network and the network so obtained at the last batch are returned as output value

- **ReduceExemplars:** this function take as input the list of exemplars and the parameter m, that is obtained as $m = \frac{K}{T}$, where T is the number of classes knew at the current task. The method cut the tail of each list of exemplars indexes from the m^{th} value to the last one. It is implemented in this way because Exemplars are inserted in the list ordered: the first exemplar is the most important, the most representative of its class.

- **GenerateNewExemplars:** in order to generate the m exemplars for the new classes, the function select, for each class, the list of indexes of image from the original dataset that belong to the class. Each image from this list is passed on the network and the resulting features are stored. When all the images are been processed, the mean of all the features is calculated and normalized. Then a mean of all the images by features are computed; Giving this two elements (mean μ and features of all the images of class Y), it iterates for 0 to m until all the exemplars are collected. For each iteration, in order to select an image as exemplars, the model compute for each image x the mean $\mu_{ex}(x)$ of all the exemplars and the image itself; then, it select the exemplar for which $\mu_{ex}(x_i)$ is closer to μ. Each time an exemplars has been selected, it is removed by the list of indexes to be sure that there are not duplicated exemplars. The model returns the updated list of exemplars.

6.2.3 Classifier

For every image in the analyzed classes, to execute the classification the model uses a special method, called "classify" that uses a Nearest Mean-of-Exemplars classification. First, the method stores the features extracted from the images passed as input in a matrix. So, the matrix have dimension $len(batch) \times numFeatures$ Then, for every class analyzed until the current task, it calculate the mean for the exemplars. Finally, for each element of the input data batch, it will categorized as the classes for which its features are closer to the mean of features of the exemplars saved for the class itself.

6.3. Results

The *Figure 7* represent the accuracy for ICaRL over all the steps. The last steps has an accuracy of 46%. The improvement w.r.t finetuning and LWF is large. The idea of exemplars seems to works very good, and it allow to double the accuracy at the last step. The model is now able to predict the correct label for elements of all the classes, even if not perfectly: it know about the existence of the 100 classes, but weights and bias are more sensible to new classes.

To better understand the results obtained and the weak points in ICaRL, we will watch the results of some variations:

- If instead of taking herding exemplars, the model selects them randomly, the accuracy at the last task is two points higher than the original model.

- If we extend the number of stored exemplars, by removing the cutting operation in each task, the accuracy increase of four points with respect on ICaRL implementation with a fixed number of exemplars.

So we have highlighted that one problem is the imbalanced dataset: the exemplars are not enough in number to assure a perfectly balanced model.

Another aspect to take into consideration, more difficult to prove with an experiment but that is intuitive, is that increasing the number of classes, it's more probable to find classes that are similar to one each other and that are represented by similar features. So it is more probable to have mislabelled classes.

We can see evidence of the bias in Figure *Figure 8*, where it's clear that there is a very imbalanced predict phase in the model: the horizontal line defines the true count of the presence of each class, i.e. how many times each class is present in the analyzed dataset. The last classes analyzed have been assigned several times the average, but we can see that many classes do not reach half the average value.

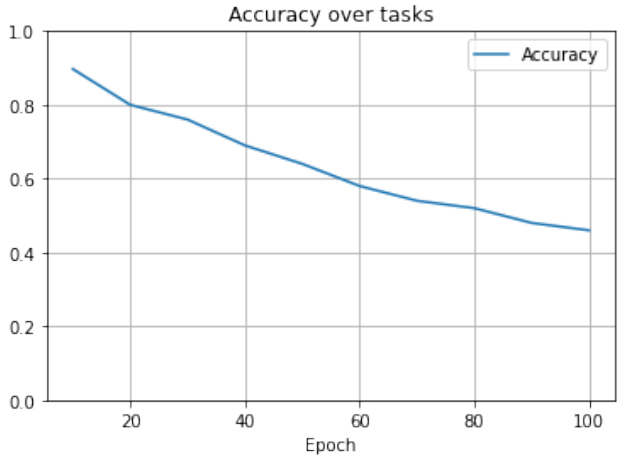

Figure 7. Accuracy ICaRL

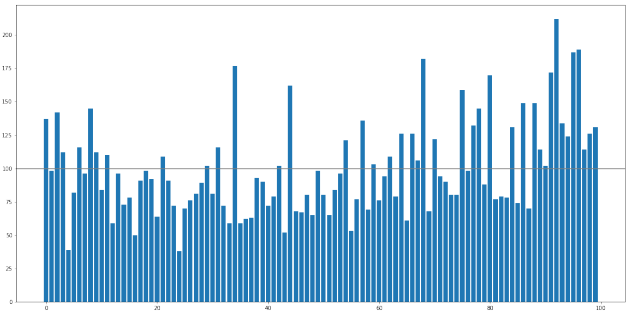

Figure 8. Histogram of predicted labels in ICaRL

7. Different Losses

Let's look at some implementations that goes deep into ICaRL possible improvements.

7.1. Mean Square Error Loss

Given an element x and a target y, the mean square error measures:
$l(x, y) = mean(L), L = \{l_1, ..., l_N\}, l_n = (x_n - y_n)^2$
It's a standard loss for classification problem. We have tried to apply it both for classification and distillation loss. The approach is the same used for the Binary Cross Entropy Loss, but the accuracy at the last step is lower than the one obtained by BCE.

7.2. Cross Entropy and Mean Square Error Loss

A different approach can be to use different losses for Classification and Distillation phases: to obtain the classification loss a Cross Entropy is used (it takes as input the labels and not their one-hot version), while for distillation a MSE is used. The two losses are calculated in the reduction mode and then are summed. But this approach give a very bad result: the accuracy at the last task is under 40%.

7.3. Cross Entropy and Cosine Loss

In this case, a new loss is used for distillation phase. The cosine similarity between two non-zero vectors is a measure of the angle between them: $loss(x, y) = 1 - \cos(x_1, x_2)$ where x_1 is the output of the trained network for older classes and x_2 is the output of the freezed network for the same classes. This combination is the one that performs better over the three new losses experimented: the accuracy at the last layer is of 42%

8. Different Classifiers

In this section we would like to check the network response to different classifiers and to see if they can lead to further improvement.

8.1. K-Nearest Neighbors classifier

The first classifier proposed is the KNN. We decided to use that because it was the having that in a certain way echoed the one of the classifier used in the paper. KNN is a classifier that has the aim to find the k elements of the dataset that have the closest characteristics with respect to the given element. Once the elements are found their label (or the label belonging to the majority of them) is assigned to the element.
We used the API provided by the `Scikit-Learn` library We found the best value of K to be **K=3**. By applying the model we can notice the following results:

$$[0.85, 0.72, 0.57, 0.41, 0.26, 0.17, 0.13, 0.10, 0.085, 0.06]$$

That are low in terms of test accuracy compared to the NME classifier. For this reason we decided to try another classifier.

8.2. Linear SVM classifier

The second proposed classifier is the linear SVM classifier.
The Linear SVM is a classifier based on support vector machines that has the aim to separate elements belonging to different classes through lines and to assign a label to new elements according to the distance from the lines separating the classes. As before we apply the model by using the API provided by the `Scikit-Learn` library. As a result we obtain the following values of test accuracy :

$$0.83, 0.69, 0.39, 0.29, 0.21, 0.19, 0.11, 0.11, 0.10, 0.10$$

The results shows that the classifier has low performances in terms of accuracy compared to the NME classifier.

8.3. RBF kernel SVM classifier

In this section we try to increase the performances obtained with the previous classifier by substituting it with a RBF kernel based SVM. The RBF kernel support vector machine is a SVM that uses an RBF kernel instead of a linear one to separate elements belonging to different classes. From the implementation computed by using **C=1** as an hyperparameters we can find results that are comparable to the KNN classifier and slightly higher than the one obtained by using Linear SVM. These are the test accuracies obtained:

$$[0.87, 0.74, 0.77, 0.49, 0.37, 0.28, 0.24, 0.24, 0.23, 0.21]$$

We can see that this classifier is the one that better performs among three even if is still not able to reach the results obtained by the paper.

9. Proposed Improvement

In this section, we want to analyze the weaknesses of the ICaRL. By carrying out the analysis, we found that the problems in the model could be caused by one of the following:

- the number of exemplars is the same among all the classes, but the net forgets the classes seen at the beginning more often than the last seen classes;
- using the average of the features for the classifier is a good idea, but if some classes have outliers, they will probably be classified incorrectly;
- the model is penalized by the high presence of similar classes, which probably require similar characteristics to be described. This can cause information loss for older classes;

To try to solve these problems, we proposed some changes to the ICaRL algorithm and analyzed the results.

9.1. Increment Exemplars

The dataset is unbalanced because the number of exemplars saved from previous classes is lower than the number of images available for the current ones. But this is needed to satisfy one of the constraints of incremental learning: the old data are not available while the model is learning new tasks.

We can see how the model performs if we select a higher number of exemplars, e.g. if the model avoids the reduce-exemplars method.

The higher performances obtained by using this approach suggest that find a solution to augment the number of exemplars, without waste memory, could be a good improvement. So, to simulate a generator of images, we have tried to apply a stronger data augmentation to the exemplars,

during the training phase. The model works as described next: the method Subset implemented in class "DataSet" has been adjusted with an if-condition that, if the image analyzed belongs to the list of exemplars, apply a particular transformer; else, apply the standard one. By using this approach, we expected to have a little increase in model accuracy. The data augmentation applied as explained, overall, didn't change the performances. A possible solution to this problem can be to use a network that generates new artificial images based on the exemplars stored.

9.2. Hyper-rectangles classifier

Exemplars-based learning is a recent addition to incremental learning literature. There exist many "families" of algorithms based on this concept, one is the NME; it can be classified as a "instance-based learning": it stores exemplars as points and never changes it. The only decision that influences this methods are which exemplars store and how calculate the distances.

A generalization of the concept of nearest exemplars is the *Nested Generalized Exemplars learning algorithm*. It look at the images like objects in an Euclidean $n - space$. Each object is an hyper-rectangle in a space of n dimensions, where n is the number of features of the object itself. The version implemented in this paper has some important differences with respect the one presented in paper [5]:

- The initialization of the exemplars space is not random, but the hyper-rectangles are created starting from the selected exemplars during the training-phase: for each class seen at the current task, the model create an hyper-rectangle that contains (in a spacial meaning) all the exemplars selected. Because the modality of selection (exemplars closest to the mean are selected) this implementation should cover enough space to be the best approximation for all the images of the class.
- No new hyper-rectangles are created during the classification phase; that because the rectangles created during the initialization phase represent all the classes available, so there is not possibility of create a new class if it is not in the exemplars set.

So, the algorithm works as follow:

1. For each class in analysis, the list of all the exemplars is taken.
2. Given the list of exemplars, by using the features extractor of ICaRL and by comparing each exemplar to each other, the maximum and the minimum value for each feature are find out. By using these values, an hyper-rectangle is constructed. The hyper-rectangle closes the space of all the features that defines the exemplars of that specific class.
3. For each image in the test set that the model have to classify, it extracts the features and compute the euclidean distance of each image from all the hyper-rectangles. The image receives the label of the closest rectangle or, if two rectangles have the same distances, the label of the smaller one (when smaller is referred to the volume of the hyper-rectangel).
4. If the image is into the space defined by the rectangle (i.e. that the distance is 0), nothing append. Else, if the classification is correct [*Predicted label = True label*] the rectangle is update in order to include the new element.

The classifier described so far has some weakness, and the performances obtained using it are not so good, especially the low accuracy at the first batch suggest that the model can be improved. Possible future improvements can be to use, instead of the maximum and minimum value for each features, the value of $\mu \pm \sigma$, where μ is the average value and σ is the standard deviation, to construct the hyperspace; to avoid nested rectangles and to use a more sophisticated metric to calculate the distance

9.3. Bias Correction Layer

As reported in *Figure 8*, the model is imbalanced to the last classes seen; a possible solution to this problem could be to manually re-balance the classification. In literature is available a solution to balance the fully connected layer: the *Bias Correction Layer* [2]. It is an additive layer, added after the last fully connected, and is a linear model with two parameters [α and β]. For each received element, it applies the following operation:

$$x = \alpha \cdot x + \beta$$

In order to train the BIC layer, and learn the best parameters, at each step the training set is divided into train and validation set, for both new classes (Validation Set of new classes) and exemplars (Validation Set of old classes) : the training set is used to learn the convolutional and fully connected layers of ICaRL, while the validation sets are used to learn α and β.

At each $task \geq 0$, the model call the *stage2* function that iterates over the two validation sets, that must have the same length, and at each batch retrieve the output of the ICaRL last fully connected layer for the images and, by computing the MSE loss, compare the outputs of the network for the new classes with the one for the older classes. By this comparison, it estimates the bias and adapt the parameters. Then, during the test phase, the part of the output that corresponds to the last classes is passed through the BIC in order to remove the bias. The performances of this implementation are lower than the one obtained by ICaRL, but it is a good results because:

- With a deeper fine tuning, we could find a solution that performs like ICaRL
- It suggest that the problem in ICaRl is not the last fully connected layer, but probably the problem is in the convolutional layers that are not able to correct represent the features spaces of all the classes. This can be solved by applying some kind of distillation to intermediate feature space. [4]

10. Model results comparison

The *Table 2* contains, for all the methods presented in this paper, an overview of the results: the left-column contains the list of the accuracies over all the tasks; the right-column contain the delta of accuracy between the first and the last batch

LWF	
Accuracies	*Delta*
0.89, 0.0.72,0.63,0.51,0.44,0.39,0.32,0.30,0.26,0.24	0.65
I-Carl	
Accuracies	*Delta*
0.89, 0.80,0.74,0.68,0.64,0.59,0.54,0.51,0.48,0.46	0.43
CE+Cosine Loss	
Accuracies	*Delta*
0.88, 0.81,0.75,0.66,0.62,0.55,0.51,0.49,0.45,0.43	0.45
CE+MSE Loss	
Accuracies	*Delta*
0.89, 0.72,0.62,0.55,0.49,0.44,0.42,0.40,0.37,0.35	0.54
MSE+MSE Loss	
Accuracies	*Delta*
0.88, 0.79,0.75,0.69,0.62,0.58,0.52,0.49,0.44,0.40	0.48
KNN Classifier	
Accuracies	*Delta*
0.85,0.72,0.57,0.41,0.26,0.17,0.13,0.10,0.08,0.06	0.79
Linear SVM	
Accuracies	*Delta*
0.83,0.69,0.39,0.29,0.21,0.19,0.11,0.11,0.10,0.10	0.73
RBF Kernel SVM	
Accuracies	*Delta*
0.87,0.74,0.77,0.49,0.37,0.28,0.24 ,0.24,0.23,0.21	0.66
Increment Exemplars	
Accuracies	*Delta*
0.90, 0.80,0.77,0.71,0.66,0.61,0.58,0.56,0.52,0.50	0.40
Hyper-Rectangles Classifier	
Accuracies	*Delta*
0.75,0.41,0.36,0.32,0.31,0.28,0.26,0.24,0.23,0.21	0.54
Bias Correction Layer	
Accuracies	*Delta*
0.88, 0.81,0.75,0.66,0.62,0.55,0.84,0.49,0.45,0.43	0.45

Table 2. Results

References

[1] R. et al. icarl: Incremental classifier and representation learning. 2016.

[2] W. et al. Large scale incremental learning. 2019.

[3] Z. et al. Learning without forgetting. 2016.

[4] Z. Michieli. Knowledge distillation for incremental learning in semantic segmentation. 2020.

[5] Salzberg. A nearest hyper-rectangle learning method. 1991.