Incremental Learning Project

Fabrizio Lande
Politecnico di Torino
s276334@studenti.polito.it

Davide Napolitano
Politecnico di Torino
s270005@studenti.polito.it

Vittorio Pipoli
Politecnico di Torino
s280139@studenti.polito.it

Abstract

The goal of this project is to recreate three incremental learning models, namely fine-tuning, LWF (Learning Without Forgetting) and iCaRL (Incremental Classifier and Representation Learning) as they have been presented in the iCaRL paper [5], try them out, see if the results on the paper are actually reasonable, find their weak points and then improve upon them by making our own take on the models in order to create a new and hopefully better performing one.

1. Introduction

Incremental learning is an interesting and relatively new field of research in the machine learning environment that tries to create classification models capable of learning to classify new classes as they arrive, without forgetting how to classify the previously learned ones. This effect of forgetting old informations, a.k.a. "catastrophic forgetting", is the major problem that incremental learning has to deal with. This is indeed an open problem with each year different takes on how to solve it or make it less relevant.

2. Data Exploration

As specified on the project pdf, we developed our studies on Cifar100 dataset [2] just like it has been done on the original iCaRL paper. Cifar100 is a dataset composed of 60 thousands images of 100 classes, already divided in a 50 thousand images for the training set and the remaining 10 thousands for the test set. It is well balanced in terms of proportions between images: each class of Cifar100 has 600 samples (500 for training and 100 for test). To make its use simpler, we created a class that automatically gives us train and test sets and randomizes the order of the classes by means of a random seed. However, because it was much simpler in terms of implementation, the class automatically creates a dictionary to map these shuffled classes into increasing labels ones (from 0 to 99) in order to reuse some previously implemented code.

3. Model Implementations

The whole implementation has been carried on the Colab environment, setting up some libraries in the code in order to use notebook scripts across other notebooks. Everything has been coded in python heavily depending on PyTorch library for exploiting GPU capabilities. Also we did rely on previously implemented code that we found online [6][3] but we would like to underline that everything has been heavily adapted and modified in order to meet our needs: most of the original code is in fact outdated or simply doesn't work as it is. We have changed, erased and rewritten many lines (even core lines) of the original codes to make it work. Regarding how we plotted, we used data from three different runs of each model in order to show some of the variation of results from run to run. We couldn't collect more than that due to time and resources limitations: Colab keeps reducing resources you are allowed to use the more you use them and, even by adding Google accounts, we quickly ran out of memory or even got locked out for a couple days from the platform. All the models presented are trained with the hyperparameters of the ICarl paper [5] (LR: 2, Weight Decay: 0.00001, Step sizes at the 49 and at 63 epoch, Gamma: 0.2, Momentum: 0.9, Number of epochs: 70 and Batch size at 128) and by using a specific ResNet optimized for this dataset [1]. Our codes and implementations can be found in the following link: https://github.com/DavideNapolitano/Project-IL.

3.1. Fine Tuning

Fine tuning is a basic strategy for incremental learning that is used at this point more as a baseline for other models than for actual usage. The idea of fine tuning is to have a common convolutional structure that is used to vectorize the input images. As proposed in [4], in the original idea each time new classes arrive a new FC is trained, the previous ones are freezed and the convolutional portion of the model is fine-tuned to adapt it to these new images and to work with this new FC, which is completely trained from scratch. However our implementation follows the one done in the ICaRl paper [5], so the FC is set from the beginning to have 100 output nodes and at each iteration a new group of ten class of images is trained and deals with the 10 nodes associated. Classification is done by means of softmax of the outputs of the network: the most probable label is assigned to the image.

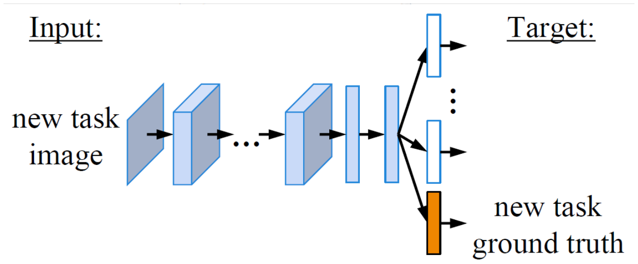

Figure 1. Fine Tuning model illustration

The model has been implemented as presented on the iCaRL paper [5] using the very same hyperparameters and techniques. Indeed the result is the same as shown on the paper (page 7):

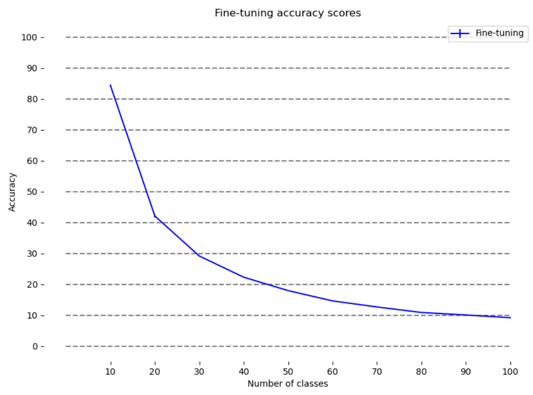

Figure 2. Fine Tuning accuracy plot

As shown in the figure above [Figure 2], it is no surprise that the accuracy variance at each iteration, defined from now on as a new batch of images belonging to ten never seen before classes, is actually very small. This behaviour can be explained by the fact that the model performs poorly on old classes at each iteration and classifies almost in the same way whichever batch of new classes it receives. This also explains the huge drop in performances that the model faces at the second iteration, which is the very first one in which it actually performs "incremental learning", going from 85 percent to 42 percent of accuracy. Furthermore, the confusion matrix for this model, in line with the one presented on the paper [5], gives us even more proofs of this behaviour:

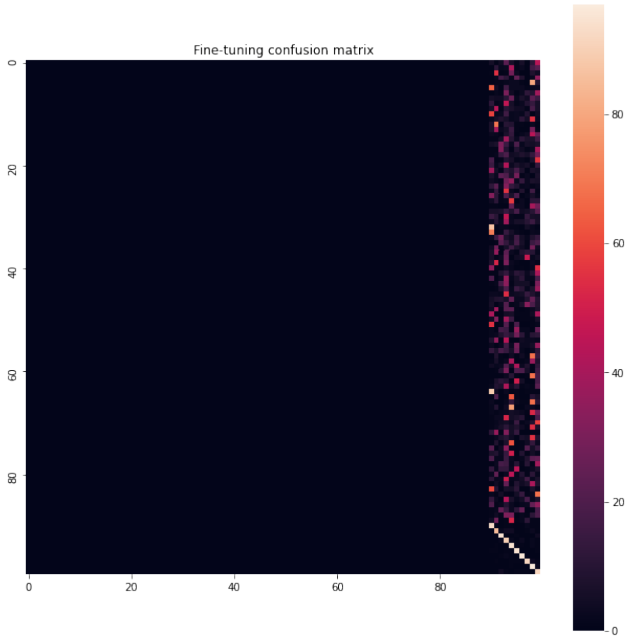

Figure 3. Fine Tuning confusion matrix

Classes from 0 to 89 are not recognized at all and only the last 10 classes, the last ones the model has learned, are well classified as we can see from that white segment on the main diagonal of the matrix. However, the fuzziness of the colored squares on the right hand side shows that on the last ten classes there is a lot of misclassification which is caused by the fact that the vectorization is not that good and therefore the nodes associated with old classes misunderstand it. Training with new images has the effect of tuning the whole model to the last ten classes only. This means that at each iteration the model will only be able to classify those classes while losing the ability of doing the same for old ones. This happens because the model tries to fit images of the last classes and changes the convolutional and FC layers accordingly, making them unable to do anything else other than classifying the last 10 classes. Another issue with this model is that the loss function does not consider the "distillation" part of it meaning that the model does not consider misclassifications on old classes and in practice does nothing to contrast the catastrophic forgetting.

3.2. Learning Without Forgetting

Learning without Forgetting is a 2016 paper [4] (later updated in early 2017) that tries to overcome the catastrophic forgetting issue by improving upon the fine-tuning idea: at each iteration the model adds new fully connected layers as they are needed. As in the Fine Tuning model, our implementation starts with the number of nodes of the FC already set to 100.

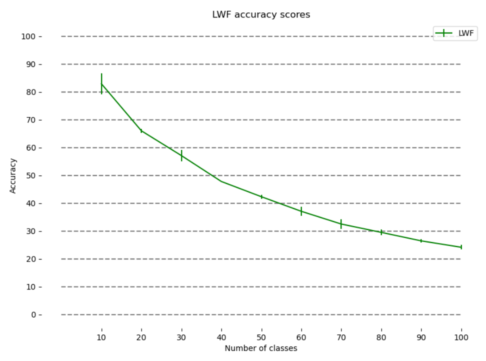

Figure 4. LWF accuracy plot

The difference from fine tuning is that the model has to ”keep in mind” its state of the previous iteration: whenever new classes arrive it cannot overfit on them too much or else it would become too different from its previous form. To do so, the model introduces the concept of distillation loss: it is computed just like the classification loss with BCEwithLogitLoss function, and calculates the difference between the output of the previous model (generated in the previous iteration) and the new one concerning already learned classes only. With this added contribute to the total loss, the model is able to learn new classes but also reduce the forgetting of old ones or else the distillation loss associated to this behaviour would grow too much. The implementation of this model is based upon a GIT repository [6] that claimed to have implemented the LWF model as explained in its original paper [4]. Because it didn't use Cifar100 as dataset and had different criteria for calculating the loss we modified it and by using the hyper parameters found in the iCaRL paper [5] we managed to find the very same results as presented in the graph (iCaRL paper [5], page 7).

We noticed that this model's performance does depend on the order in which classes are forwarded: different runs had scores that differed even by 3-4 percentage points meaning that even if on average it performs better than fine-tuning, it is not that stable as is shown by means of standard deviation bars. This improvement in performance can also be shown by means of its confusion matrix that clearly shows how old classes are still recognized and in some cases well classified.

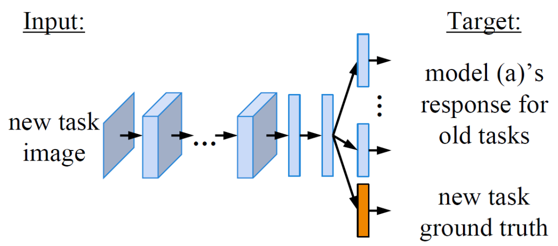

Figure 5. LWF model illustration

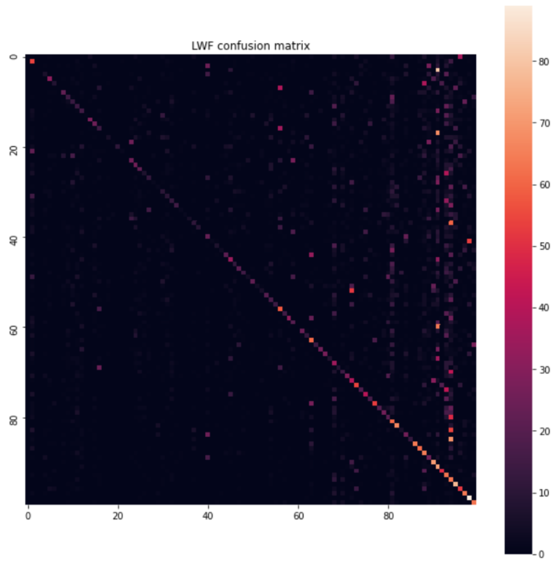

Figure 6. LWF confusion matrix

An ideal case would have the diagonal squares completely white and the rest completely black. As we can see the diagonal goes lighter and lighter the more we move towards the last classes the model has learned. Comparing it to the confusion matrix associated with fine-tuning this one clearly shows a huge improvement but it also highlights that the model is still heavily forgetting older classes almost linearly the more it learns new ones. One weak point that we thought of, which is in fact partially dealt with by its successor iCaRL, is the fact that the model is not allowed to review some samples of old classes.

3.3. ICaRl

iCaRL is a late 2016 paper (later reviewed in 2017) that tries to improve upon LWF results by completely changing the way the model classifies and how it stores informations of previous knowledge in order to fight catastrophic forgetting. The big take on the problem given by this paper is the concept of exemplars. An exemplar is an image among the ones forwarded to the model at training time that is close enough to the ”mean image”, an ”image” whose vectorized representation is the mean vector of all the other images of that class. Basically at each iteration the convolutional layers generate a vectorization of the input images based on the current state of the model at that time. For each class we compute the mean image vector and then we pick m images whose mean is the closest to the actual one, we sort them by ascending closed distance. These selected images become exemplars. The model saves the raw image (in our case we choose to save the indexes, but for simplicity we'll refer to them always with image) and not the vectorized representation of these exemplars so that it does not depend on the current state of the model: if new classes arrive the convolutional layers are changed accordingly and so the new vectorization will be different from the previous one with respect to the same input image. These raw images are added to the new images of the new never seen before classes of the training set so that the model can still "review" them. For classification, the model vectorizes the exemplars of each class, computes the mean vector and then assigns the label of the closest mean vector to the unlabelled input image. These exemplars make the model work way better than LWF at each iteration as can be seen on the accuracy graph and on its confusion matrix:

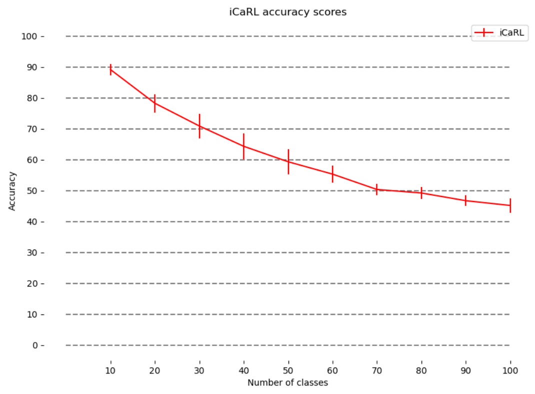

Figure 7. ICaRl accuracy plot

iCaRL is by far the model with high variance on its accuracy scores meaning that for it, the order in which it receives the new classes is very important and drastically change the performances.

Non diagonal squares are darker than LWF confusion matrix meaning that iCaRL makes less misclassifications while the diagonal has almost the same color end to end. This means that the model is dealing with catastrophic forgetting way better than LWF or fine-tuning models.

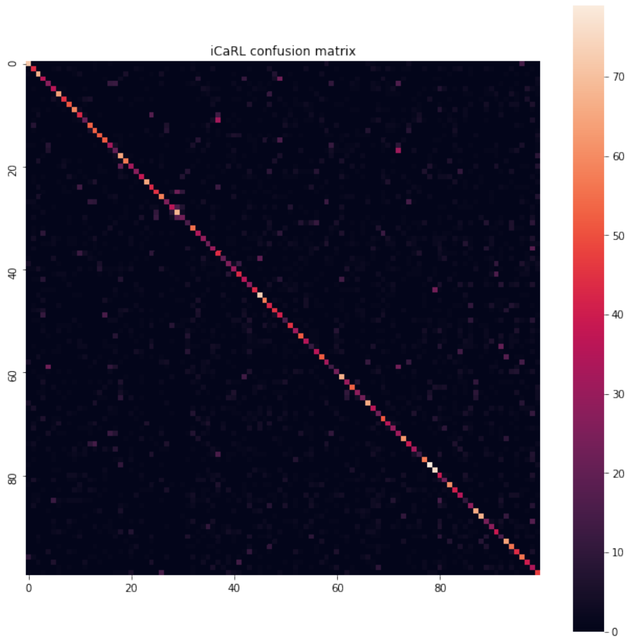

Figure 8. ICaRl confusion matrix

One point that the paper underlines is the number of exemplar images that the model can use. The original work revolves around the idea that the model cannot have unlimited memory space to save all images as exemplars but that it has to save just a small amount. The total number of exemplar is indeed fixed (in our runs and on the paper it is equal to 2000) and each class can use up to 2000 divided by the number of known classes. We thought that fixed number of exemplars for each class is a bit restrictive for the model: we think that classes that have performed worse (that have a lower precision) should have more exemplars because clearly the model is not able to learn them that well and the opposite goes for "easier" classes. Finally we want to show a graph similar to the one presented in the iCaRL paper that clearly compares the three models accuracy scores at each iteration:

As we can see iCaRL outstands the other two reaching a final accuracy of around 45-50

3.4. Note on ICaRl implementation

While we were developing our code for iCaRL, professor's assistant Fabio Cermelli suggested a "trick" to increase model performances in order to obtain similar accuracies to the ones represented on the paper. The idea that he suggested was to use all the training data of the ten new classes of each iteration in order to compute a true mean image for each class instead of computing it using exemplars images only like is done for older classes. Even though our model was complete and running at that point we gave it a try and implemented his suggestion. Here is the accu-

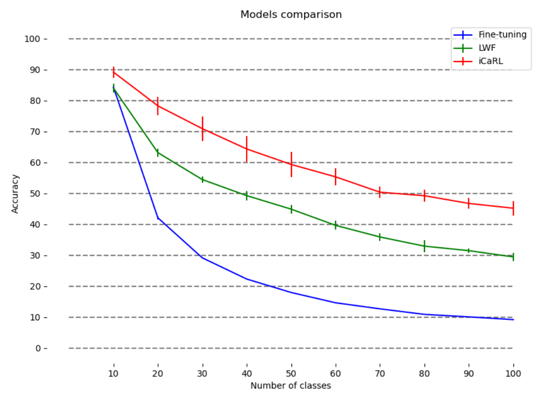

Figure 9. Comparison between the accuracy plots of the 3 models

racy comparison (the graph has been clamped between 90 percent and 40 percent of accuracy in order to enhance differences):

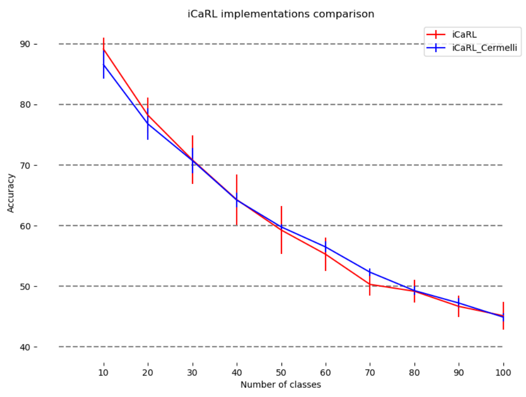

Figure 10. Comparison between the standard ICaRl implementation for the classify and the ICaRl that computes the mean of the current batch train using all the train images instead of the exemplars

As we can see the two models perform in a very similar way. In both cases the variance is so high that makes them indistinguishable from a statistical point of view. We decided to complete the following points of the project with our own implementation since there is no real difference and the one that we created is actually the one that the paper explains (we did not find any evidence on the paper that the original implementation used this "trick").

4. Tests with different loss functions

In order to understand a bit more how iCaRL worked, we tested it changing the loss functions used for calculating both classification and distillation losses. We executed three tests with three different combinations of loss functions:

- MSE-MSE
- BCE-L1 (+ BCE-L1 "cheat")
- CE-BCE

We want to specify that we did not perform major hyper-parameters tuning aside from the third model (CE-BCE). The first two work with the very same values used for the standard model (BCE-BCE) while the last one diverges with a learning rate of 2.0 so we set it to 0.2. Also, we computed only one run of each model for this point of the project because the main goal is to show a trend, and not giving informations on variance of results.

4.1. MSE-MSE

We chose mean square error as a starting point because it is a commonly used loss function which has the advantage of being easily interpretable. The classification part computes the euclidean distance between softmax of the output associated to the new classes and onehot encoding of the groundtruth. The distillation works in the same way but it compares the output of the model of the previous iteration and the output of the new one regarding old classes only.

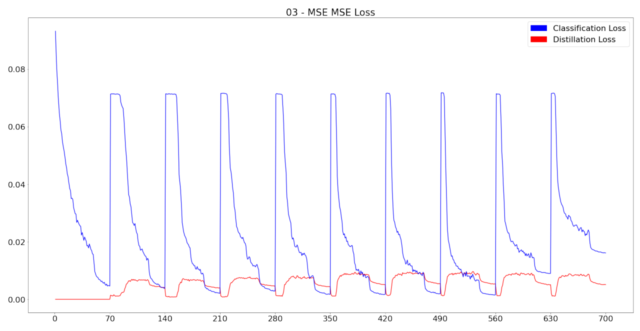

Figure 11. MSE-MSE Trend of the losses during the whole training of the model

During the first iteration (from epoch 0 to 69) only the classification loss gives a contribute while the distillation remains equal to 0. This is the correct behaviour as it highlights that the latter loss is indeed generated by comparing old classes labels only, which in the first iteration are not present. As we will show in later graphs this behaviour is consistent with any combination of loss functions demonstrating that the model is working as expected regarding the loss functions used. After the first iteration, whenever new classes are presented the classification loss goes up no higher than 0.073 and then it converges to a low value within 70 epochs. It is worth to notice that with this combination of loss functions the distillation loss gives more contribute than the classification one after 40-45 epochs. As we will show in the accuracy graph this is an important behaviour that we have to enhance to push the model not to forget old classes and so to make it work better. Last two iterations don't converge within 70 epochs probably because the convolutional layer is becoming too generic and is less and less able to learn new images and so the classification loss stays high. Note: while running each experiment we also looked at training accuracy (accuracy of the model on the last ten classes of the training set) to see if the model was actually learning or not. At each iteration the train accuracy was smaller and smaller and in the last two reached the lowest scores demonstrating our theory for explaining this behaviour.

4.2. BCE-L1

On this other experiment we tested how the binary cross entropy function performed as classification loss while we kept on trying another easy to interpret loss for the distillation part.

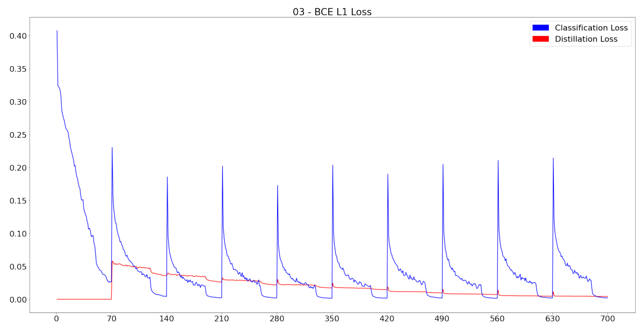

Figure 12. BCE-L1 Trend of the losses during the whole training of the model

L1 makes the distillation loss decrease as the iterations increase. A more significant distillation loss should increase as the iterations go by because old classes "are getting older" and so the model is forgetting them. We tried to explain this strange behaviour by considering these facts:

- BCE as classification loss makes the model learn the new classes very well
- The model starts with 100 nodes associated with the last layer
- Output nodes are passed to a sigmoid function (mandatory for using BCE) that returns values very close to either 0 or 1.

The ideal output of our model should be a vector of 100 dimensions where only one value is 1 and the others 0. The "active node" position is the label that the model assigns to the forwarded image. By the fact that we are using a sigmoid function on both results of the old model and the new one, when we compute their difference using L1, we get very small values in return and even though they get summed for how many known classes we have they are smaller and smaller. In the first iterations learning new classes so well makes the model perform better than other configurations but this first advantage is a disadvantage from the sixth iteration on, where the model sees a drop in performances as can be shown in the comparison graph. Note: we also implemented a modified version of this loss function (referred as cheat bce l1) to counterbalance this decreasing trend of the distillation loss or at least make it more relevant with respect to the classification loss. To do that, we added a weight factor that multiplies the distillation term. We chose at this stage of the project a fixed value (gamma=1.5) just to see if simply increasing the distillation term by a fixed 50 percent could have some effect. This idea was merely an experiment to see if reweighting somehow the two contributes could have some benefits and we found out that it does (see comparison graph).

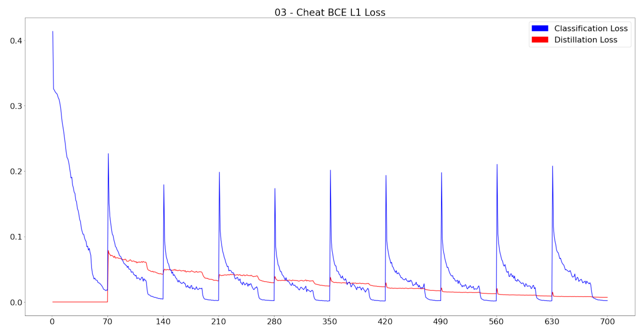

Figure 13. BCE-L1 with "cheat" Trend of the losses during the whole training of the model

4.3. CE-BCE

As last test we checked how BCE performed in computing the distillation loss while at the same time we used CE as classification loss function just to check if there were some differences with BCE.

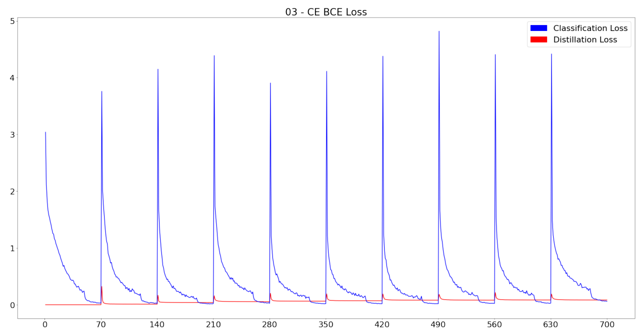

Figure 14. CE-BCE Trend of the losses during the whole training of the model

Cross entropy loss tends to stay higher than its binary counter part: aside from the first iteration, where the model in both cases has to heavily adjust itself, whenever new classes come up CE fires up to values higher than 3 (BCE topped at 0.23) and by the end of every 70th epoch, even if it tries to converge to 0 it doesn't get as low as BCE does. Regarding using BCE as distillation loss function we noticed this increasing behavior that MSE and L1 did not give. The problem with this configuration is that the distillation contribute is too low in comparison with the classification one. So even if the trend is the supposedly correct one its magnitude is so low that it is almost not relevant in the end the resulting model is the worst.

4.4. Comparisons of the Losses

Here is a graph comparing the accuracies of the different versions of iCaRL with the different loss functions in comparison with the traditional model which uses BCE-BCE:

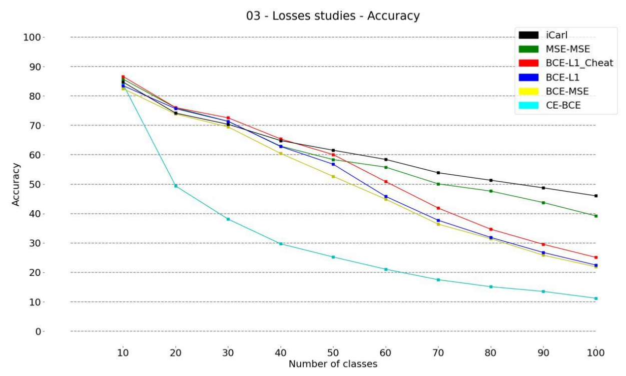

Figure 15. Trend of the accuracies provided by the different models whit the different losses

What we have learned from this experimental point of the project is the following:

- Regardless what function we use for computing the distillation term it must generate a value that is comparable to the classification loss or else it is not considered, the model does not fight against catastrophic forgetting and performs poorly. Also the opposite is important: the classification loss does not have to be overwhelming in comparison to the distillation one.

- The chosen function "has to make sense": it has to have an increasing behaviour because the more iterations the model goes through the more its contribute has to be significant.

- The classification loss doesn't have to have high spikes when new classes arrive and must be able to make the model learn new classes at a sufficiently good level.

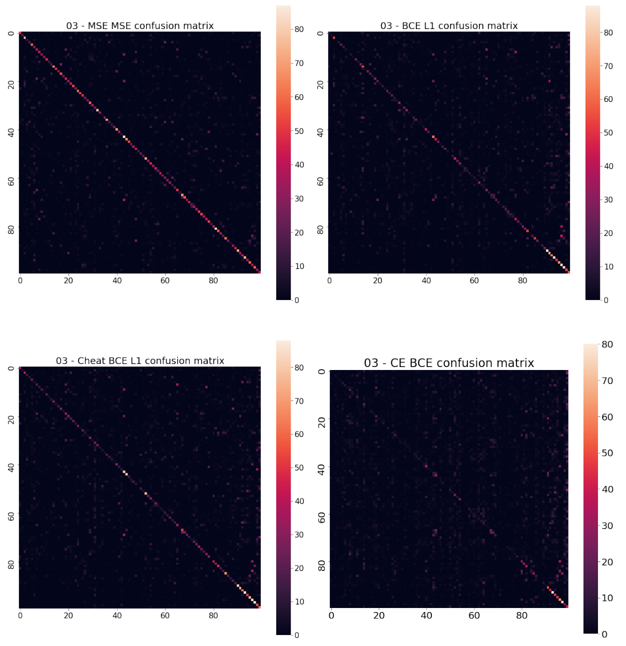

Figure 16. Comparisons between the confusion matrices obtained different models whit the different losses

- It is possible to force some of these behaviours by reweighting the two loss contributes.

BCE in practice is the best function for computing both classification and distillation loss but we think we can push its performances further by adjusting their contributes.

5. Tests with different classifiers

After making some experiments concerning the loss function, we tried to test how the model actually classifies by comparing the original classifier of iCaRL with other ones that are commonly used in the classification setting. We still maintained the concept of the exemplars but we looked at different ways on how to exploit them for the classification process. On this section we treat exemplars in a more "classification fashion": each one of them is seen as a labeled point in a 64 dimensional space (the convolutional layers create a 64 dimensional representation of the input images). Before trying our classifiers we decided to visualize these exemplars as they were created. We used t-SNE representation and after adjusting some hyperparameters we plotted the exemplars at each iterations. To accomplish this task we referred to How to Use t-SNE Effectively [7]. Here we show just some of the plots to give an idea of what we found:

As we can expect the representation becomes more and more confused as the model learns new classes. For the first 30 classes clusters are well separated, then, as the model

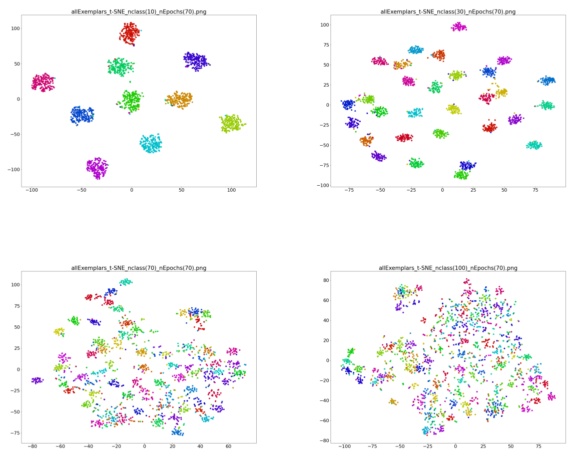

Figure 17. Clusters of the different classes: Top Left: 10 classes, Top Right: 30 classes, Bottom Left: 70 classes, Bottom Right: 100 classes

learns more, everything starts to blend together. As we have studied on our own about t-SNE the representation is that it gives us just an approximation of the real high dimensional space and it is highly dependant on the hyper-parameters that we set. Indeed we considered these plots just as a general guideline on what was going on under the hood: catastrophic forgetting actually means that the model is not able to divide old and new informations because the space in which it generates these vectorized versions of the images gets crowded and dissimilarities among classes become more shallow.

With these points in space we tried to test the following classifiers:

- KNN - k nearest neighbour: based on the t-SNE results we expected it to work that well for few classes and became poorer with too many ones but we tried anyway also to see if the visual representation made some sense. We tested some values but the differences were small. We run our tests with k equal to 10 in the end.

- SVC - support vector classifier: we did try two different kernels for it, RBF and linear, and we quickly realized that RBF was performing better and we set C = 1 after some trials.

- Random forest: we used it in order to see if decision rules were a better option for splitting the space and classify. We set number of estimators to 100 (we tried more but it performed worse).

We defined one method for each one of these classifiers and then we called them in sequence both on train and test sets. Note: instead of using the mean image for each class, so one point per class, we decided to use all the available exemplars directly. We made this choice because we thought that any classifier, aside from the original one, works better with more points. We also thought that there was no reason in collapsing all this information retained from all these exemplars into one point per class, which is something we developed even more in our own implementation. Here is the graph showing the differences in performance of each classifier: KNN as expected performs worse than the origi-

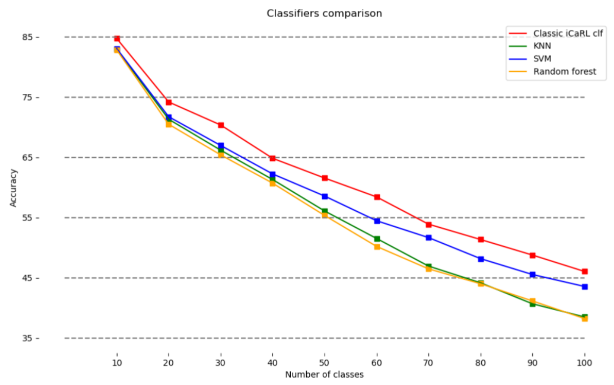

Figure 18. Trend of the accuracies provided by the different models whit the different classifiers

nal classifier due to the curse of dimensionality (we tried to mitigate this issue by using PCA in order to lower the dimensions of the space but it actually made it slightly worse or equal). Random forest is another failure in the sense that it doesn't stand the original classifier scores. This behaviour may be explained by considering once again the curse of dimensionality as possible reason but also the fact that decision rules don't work well for points generated from images since more different classes we have and way harder is to split properly the space. From our personal experience random forests in general are not the best classifiers. On some fields, such as text classification, they might work decently but we didn't expect much from them anyway. SVC gives the best results among the three classifiers. We have to point out that due to the high variance of each score we cannot take this result as groundbreaking but it still something to notice.

6. Our Model

We think that assigning a fixed number of exemplars for each class is a bit of a constraint for the model. We agree on the idea that total number of exemplars has to be fixed because in general the model cannot have infinite memory to work with. With that being said, we think that even if the number of total exemplars is fixed we can still work on how many exemplars give to each class Listing 1.

Listing 1. REWEIGHTED EXEMPLARS
```
w = icarl . compute_precisions ( y_true , y_pred , [ diz [ c ] for c in
        randomlist [ s*num_classes : s*num_classes + num_classes ] ] )
w = np.around(w,decimals=4)
mean_w = np.mean(w)

icarl . delete_last10_exemplar ()

for z,y in enumerate( randomlist [ s*num_classes : s*num_classes +
        num_classes ] ) :

    imagesInd = cifarTrain . getClassIndexes ( [ y ] )
    images    = Subset( cifarTrain , imagesInd)

    if w[z] == mean_w:
        new_m = m
    else :
        if w[z] > mean_w:
            new_m = float (m) - float (m)*w[z]
        else :
            new_m = float (m) + float (m)*w[z]

    if new_m > int(new_m) + 0.5:
        new_m = int(new_m) + 1
    icarl . construct_exemplar_set (images, new_m)
```

On any iteration, the model receives new images of ten new classes which has never seen before. These classes are learned as the model gets trained on them and then it is asked to label them: we record the precision the model has for each new class that has just classified Listing 2.

Listing 2. PRECISIONS
```
def compute_precisions ( self , y_true , y_pred , last_classes ):
    precisions = []
    for c in last_classes :
        correct = sum(y_pred[ y_true == c ] == c)
        wrong   = sum(y_pred[ y_true == c ] != c)
        tot     = correct + wrong
        prec    = correct / tot
        precisions .append(prec)
    return softmax( precisions )
```

These precisions are collected in a vector on which we perform a softmax operation. We do this so that they all sum up to 1. We compute m, number of exemplars for each class just like iCaRL did: $m = K/NumClasses$, where K is the total amount of exemplars we can save in memory. This number is multiplied by the vector of normalized precisions so that it returns 10 new values of exemplars to use for each of the 10 new classes. Normalizing the weights is mandatory or else we would not stay within the K total exemplars. Before rebalancing the exemplars of the train data, we delete them from the ICaRl model, where they have been used in the training, in order to add then the new computed ones. On the first iteration these operations are all we have to do. From the second iteration on we also need to make room for the exemplars of the new classes. To do this we simply take $1/NumIterationsPreviousExemplarsAssignedEach Class$ so that we don't interfere with the proportions calculated at training time. The way exemplars are selected both at the beginning and when cutting for other exemplars is the one implemented by iCaRL, the only thing we change is how many must be taken . We want to point out that due to numerical rounding the total number of exemplars is not ensured to be exactly equal to K but it varies by no more than +1 percent. We consider this result acceptable because an increase of that magnitude doesn't change that much the usage of memory required by the model and so the end user should not even notice it.

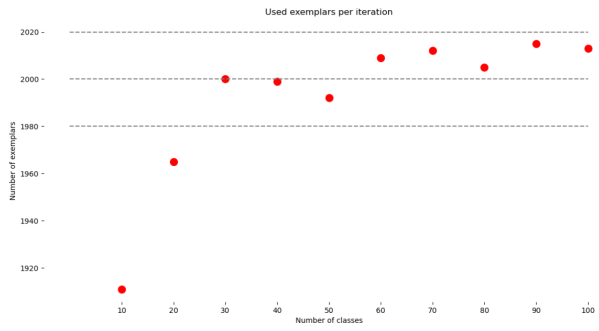

Figure 19. Exemplars reweight plot. The three dashed lines represent the ideal case (always 2000) and +-1 percent from that value.

Here is the result using this type of approach:

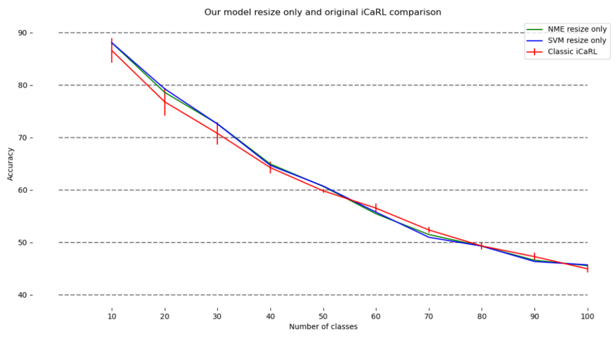

Figure 20. Comparison between our implementation with only the management of the exemplars and the standard ICaRl

Because on point 4 of the project we noticed that SVM was a good alternative to the classic NME classificator we decided to carry them both to this last final step in order to see which one performed better. We did not have found any significant difference between the two models however, neither on this experiment nor on the final model. However, regardless the type of classifier used, the model sees some noticeable improvements on the first 5 iterations and then it converges to the classic model in terms of accuracy. This behaviour can be explained by the fact that some classes might have been easy to learn at the time they have been presented but not anymore on further iterations so the model, having saved fewer exemplars for those classes, at later iterations would actually need more but it doesn't and so its scores fall. Trying to push the results even further we also worked on the way the model computes the loss Algorithm 1.

Algorithm 1 Our Loss

```
Input: $X^s...X^t \rightarrow new\ classes$
t = number of new classes
s = number of known classes
Require: $P = (P_1...P_{s-1}) \rightarrow exemplar\ sets$
Require: $\Theta \rightarrow current\ model\ parameters$

    D $\leftarrow \bigcup_{y=s...t} \{(x,y) : x \in X^y\} \cup \bigcup_{y=1...s-1} \{(x,y) : x \in P^y\}$
    for $y = 1,...,s-1$ do
        $q_i^y :\leftarrow\ g_y(x_i)\ \ for\ all\ (x_i, \cdot) \in D$
    end for

    $D \rightarrow DataLoader \rightarrow Batch \rightarrow loss$
```

$$loss_{Batch}(\Theta) = \sum_{(x_i,y_i)\in Batch} [\frac{1}{batchsize \cdot t}] \cdot$$

$$\{ [\textstyle\sum_{y=s}^{t} \delta_{y=y_i} log(g_y(x_i)) + \delta_{y\neq y_i} log(1-g_y(x_i))] \cdot$$
$$\cdot [1 + e^{(1-\frac{t}{s+t})}] +$$
$$+ [\textstyle\sum_{y=1}^{s-1} q_i^y log(g_y(x_i)) + (1-q_i^y) log(1-g_y(x_i))]\}$$

The basic idea is to mitigate the unbalance between the two contributes (classification and distillation) given by the intrinsic math of BCEwithLogitsLoss (reduction="mean"). Each term is weighted according to the number of classes it has to deal with: the classification term considers 10 classes all the time while the distillation loss works with more and more classes at each iteration. Because of this, the distillation loss becomes more and more relevant in an "artificial way". To counterbalance this effect we reweighted the classification loss to make it greater than what it actually is based on the iteration count. This weight has a decreasing trend because we still want to make the distillation loss relevant on further iterations in order to preserve knowledge of old classes.

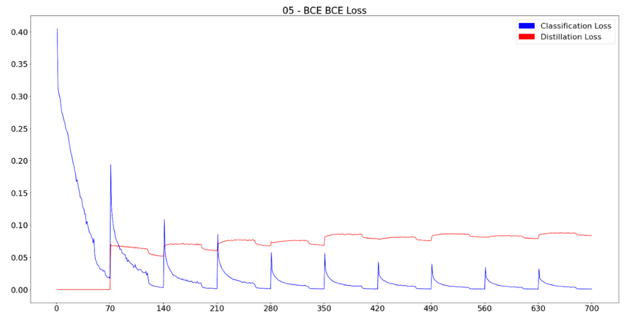

Figure 21. Trend of our losses during the whole training of the model

This modified loss makes the model perform better on each iteration (in terms of accuracy if compared to the classic iCaRL) even though with a lower and lower magnitude the more iterations we have. This effect can be expected because we are gently bringing the loss function to its original form in order to avoid a more extreme effect, similar to the one highlighted on the experiment with BCE-L1 loss: if we kept the classification loss too high, the model would have suffered of catastrophic forgetting at further iterations in the same way of the experiment.

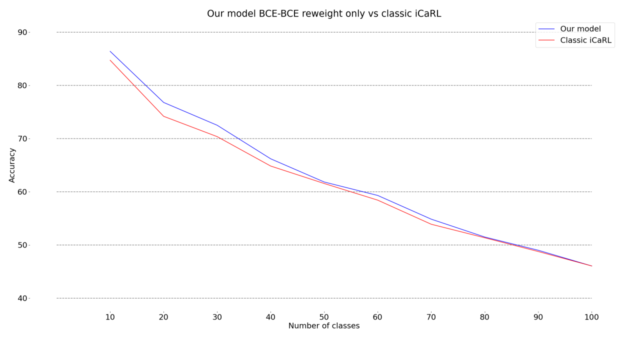

Figure 22. Comparison between our implementation with only the management of the losses and the standard ICaRl

After we tested these two ideas we decided to create a model that used them both. We got the following results:

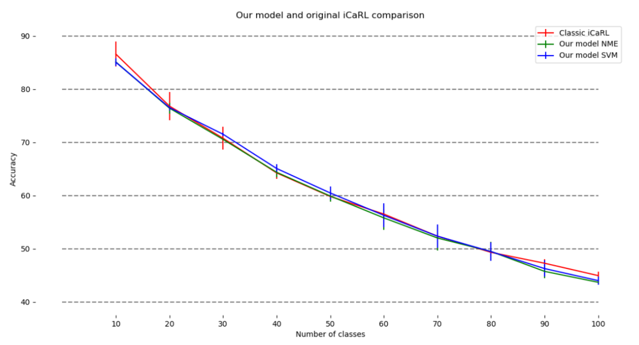

Figure 23. Comparison between our entire implementation and the standard ICaRl

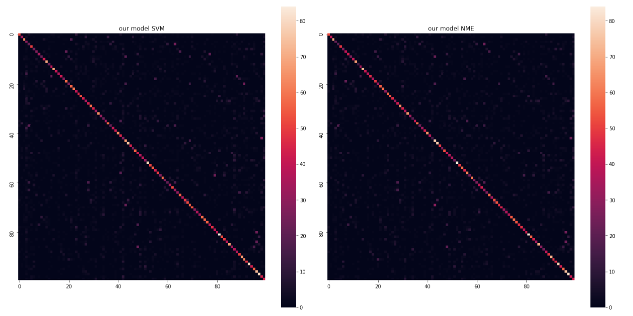

Figure 24. Comparison between the confusion matrices obtained with SVM and NME in our implementation

Unfortunately these two techniques seem to cancel each other out and the overall performance manages to be just like the original model. We also tested how our model performed if we increased the number of exemplars that it was allowed to use. We set k=3000 and we ran it again.

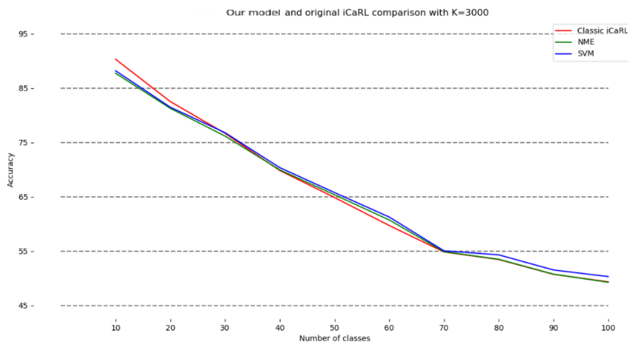

Figure 25. Comparison between our model and ICaRl with k=3000

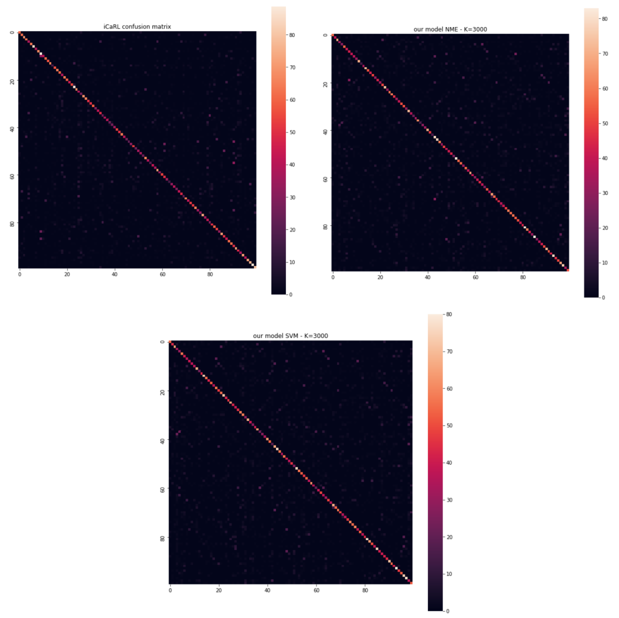

Figure 26. Comparison between the confusion matrices obtained with K=3000 with SVM, NME and standar ICaRl

With more exemplars the model ends up to be slightly better in the last iterations, which are the most important ones. This may be due the fact that the rebalance of exemplars is more significant or that the model is able to make a better use of these uneven distribution of exemplars.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. 2015. Deep Residual Learning for Image Recognition.

[2] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009. Cifar10 and Cifar100 datasets.

[3] D. Lee. icarl: A pytorch implementation of icarl, 2017.

[4] Z. Li and D. Hoiem. Learning without forgetting, 2016.

[5] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. icarl: Incremental classifier and representation learning. 2016.

[6] A. Thai. Lwf: Implementation of learning without forgetting, 2018.

[7] M. Wattenberg, F. Viégas, and I. Johnson. How to use t-sne effectively. *Distill*, 2016.