Machine Learning and Deep Learning: Incremental Learning Project

Filippo Barba (277898), Francesco Guardamagna (277388), Matteo Villosio (276054)
Politecnico di Torino
{filippo.barba, francesco.guardamagna, matteo.villosio}@studenti.polito.it

Abstract

*Learning is, by definition, incremental. It is then quite natural for machine learners to aim at having systems that can extend their knowledge over time: in the age of **IoT** and social media, the ability to learn from continuous streams of data would not only make us one step closer to natural learning but would also bring a performance increment. In this paper we follow the various steps that brought to the current state of the art, **iCaRL**, and we then try to propose some new original ideas.*

1. Introduction

Learning is, in nature, incremental: at school children build their knowledge on top of previously learnt concepts, animals do not forget an old predator when a new one appears. Given the widespread presence of continuous streams of data in modern industry and the capillary deployment of learning systems with limited power, the ability to learn in a continuous way would be greatly beneficial for present Machine Learning methods.
In this paper we followed multiple steps:

- At first, we proceeded by computing a baseline, that is, we checked the effects of catastrophic forgetting: such thing was achieved by training the network on a batch of 10 classes.
- Similarly, we computed a benchmark by using the **Joint Training strategy**, that is, by retraining the net at each step with all the data available up to this point.
- We then implemented the **Learning Without Forgetting**[3] method, the first academic step for Incremental learning. With this basic yet quite interesting strategy we saw the first improvement with respect to catastrophic forgetting.
- We then proceeded with the implementation of **iCaRL**[4], a famous strategy for Incremental Classification and Representation Learning.
- Finally, we proposed some variations to the standard **iCaRL** implementation that could be beneficial in some particular cases.

1.1. Properties of an Incremental learning algorithm

As suggested in the iCaRL paper, a class incremental algorithm should respect three properties[4]:

- it should be trainable from a stream of data in which examples of different classes occur at different times;
- it should at any time provide a competitive multi-class classifier for the classes observed until then;
- its computational requirements and memory footprint should remain bounded, or at least grow very slowly, with respect to the number of classes seen up to that point;

2. Other methods

2.1. Finetuning

Finetuning is the simplest method: it does not take advantage of any strategy to avoid catastrophic forgetting but it simply consists in training our network by finetuning on the new data we receive. Obviously, this method performs rather poorly when introducing a lot of new data incrementally, as it only focuses on classifying new data.

2.2. Learning without Forgetting

Learning without Forgetting is a knowledge preservation method which tries to prevent catastrophic forgetting of previously learnt data by introducing **Knowledge Distillation**. Differently from the **Finetuning approach**, which does not take any measure to prevent catastrophic forgetting and its solely based on finetuning the network on new data, **LwF** adds a distillation component to the learning process, allowing the conservation of information learnt during previous

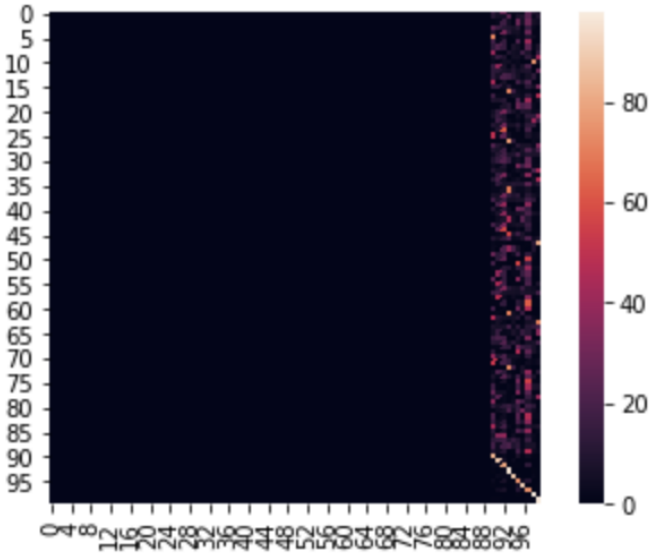

Figure 1. Confusion matrix of the finetuning method

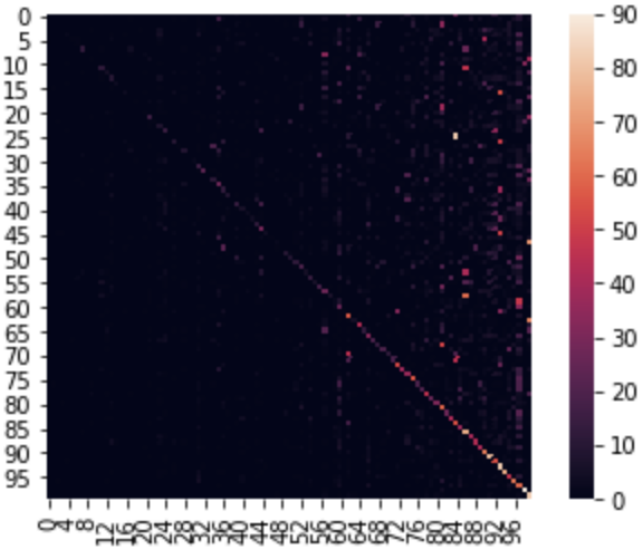

Figure 3. Confusion matrix of Learning without Forgetting method

iterations.

When new data is presented to the neural network, the **LwF** strategy first, if present, makes a prediction on the net trained at the previous step and then uses this information to add a distillation factor to the loss computation when training on new data. We do not keep any sample of the old data, and we only preserve knowledge of previously seen classes by including the contribution of the "old" network to the loss used in the training of new data, in the form of distillation loss.

It obviously performs considerably better than the **Finetuning** method, since previous knowledge is preserved in some way, but it still presents some room for improvement [*fig:2*].

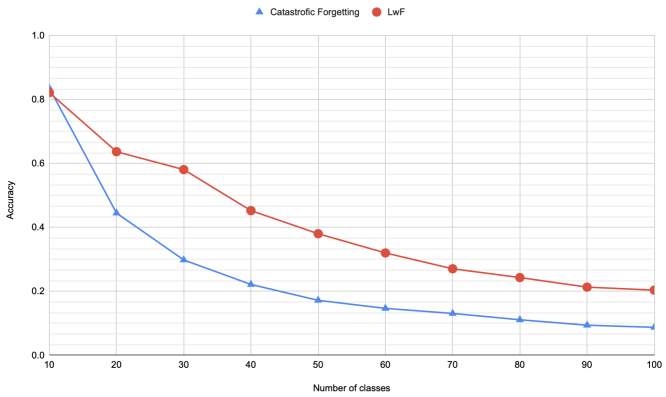

Figure 2. Accuracy trend of the LwF vs Catastrophic Forgetting

3. iCaRL

3.1. Introduction

iCaRL's goal is to incrementally classify data produced by a stream: data presented at every step are sampled from new classes and consequently, on one hand, the model should evolve in order to learn how to correctly classify the new data while, on the other, it should not forget previous classes. Moreover, when new classes are fed we don't have all the data associated with the previous classes but only a subset of that data [1] which will be referred to as **exemplars** from now on.

3.2. Our implementation

As a first step, we focused on implementing the iCaRL paper model in **PyTorch**. iCaRL uses a convolutional neural network for feature extraction with a single classification Linear layer. To further generalize our problem, the output nodes are incremented each time new data is presented to the network.

A **ResNet32** is used as backbone architecture for the implementation of **iCaRL**: such convolutional neural network became famous and widely used for its ability to overcome the issue of the vanishing gradient; thanks to the idea of *shortcuts*, connections that skip some layers, it was in fact possible to implement networks of extreme depth.

As loss function we use PyTorch's $torch.nn.BCEWithLogitsLoss$ both for classification and distillation.

Concerning **exemplars**, we strictly follow **iCaRL**'s

[1] so we have also to understand how to select the most representative samples of all the classes observed so far

strategy.

3.3. Training

iCaRL processes data from a stream, and every time data from new classes is available, the model is updated; consequently the exemplar set, containing samples of the classes observed until that point, has to be updated. The update routine, to improve the feature representation every time new classes appear, consists of three main steps:

- At first, we create an augmented training set, consisting of images of the new classes and of the exemplars of the previous classes, if present.
- After the creation of the new augmented training set, the network outputs for all the previous classes on the current network (which has not been trained for the new classes yet) are stored, using all the samples of the new training set.
- Finally, the network parameters are updated in order to minimize a loss function, consisting of two contributions. For each new image we want to predict the right label for the new classes (classification loss) and we also want to reproduce the scores stored during the previous step for the old classes (distillation loss). The distillation loss is a regularization term that prevents the loss, during the new learning step, of the information learnt previously.

3.4. Exemplar sets

As explained in the previous paragraph, every time we encounter new classes we have to update the exemplar sets. If t classes have been already encountered and the maximum number of total exemplar that can be stored is K, we have to generate one exemplar set for each class with a number of exemplars $m = \frac{K}{t}$. We follow **iCaRL**'s strategy for selecting and reducing the exemplars for each class. To create a new exemplar set, the exemplars are selected iteratively from the training samples of a new class. When generating the exemplar set for a class, we extract features from the class images using the convolutional network trained until that point. We select the exemplar which, added to the average feature vector of the already selected exemplars, best approximates the class' average feature vector; this procedure is executed until the target number of exemplars m is reached. In this way we can generate an exemplar set as a prioritized list. When introducing new classes we decrease the number of exemplars by dropping the last exemplars per class [2], in order to keep the total number of exemplars fixed to K. Since the exemplars are saved as prioritized lists, we only keep the ones which best describe each class.

[2] i.e. exemplars further from the class mean

3.5. Classification

For the classification step **iCaRL** uses a **nearest mean of exemplars** approach. To predict a label for an image we first have to compute a vector, containing for each class observed so far the average feature vector computed over all the exemplars of that class, then we compute the feature vector of the image we have to classify and finally assign the class label by computing the distance between the feature vector of the image and all the average feature vectors computed over all the exemplar sets. The image is classified as belonging to the nearest class based on the distance computed between the average features vectors over all the exemplars of that class and the features vector of the image. Similarly to the construction of the exemplars set the feature vectors are extracted using the current feature extractor of the net.

3.6. Results

Our implementation of iCaRL with the following parameters performed as seen in the *fig:6*:

$$\begin{aligned} Momentum &= 0.9 \\ WeightDecay &= 10^{-5} \\ BatchSize &= 128 \\ NumEpochs &= 70 \\ LRdrop_1 &= 49 \\ LRdrop_2 &= 63 \\ LR &= 2.0 \\ \gamma &= 0.2 \end{aligned} \tag{1}$$

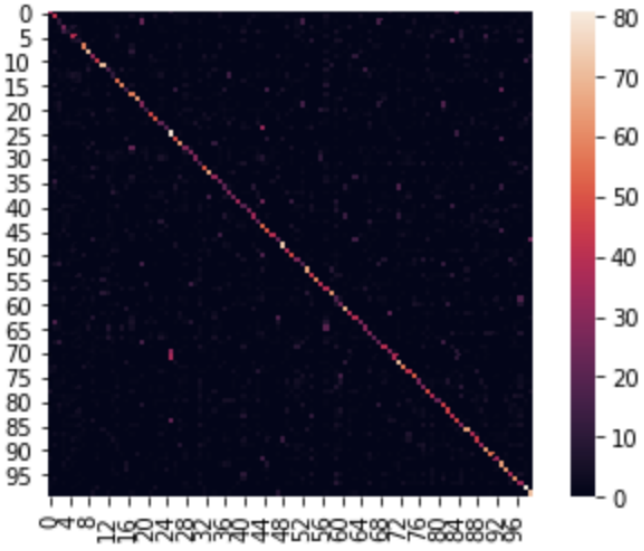

Figure 4. Confusion matrix of standard iCaRL method

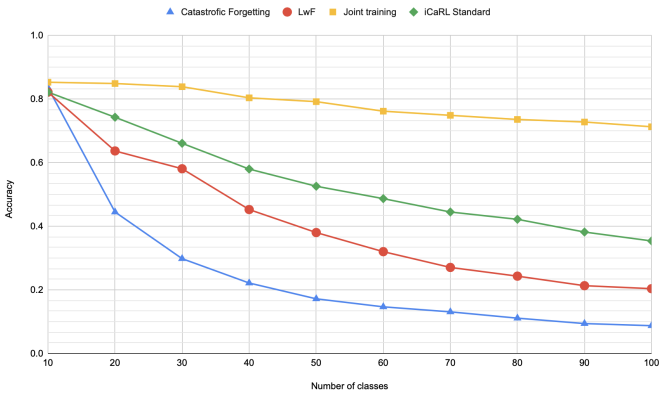

Figure 5. Accuracy trend of the iCaRL standard implementation

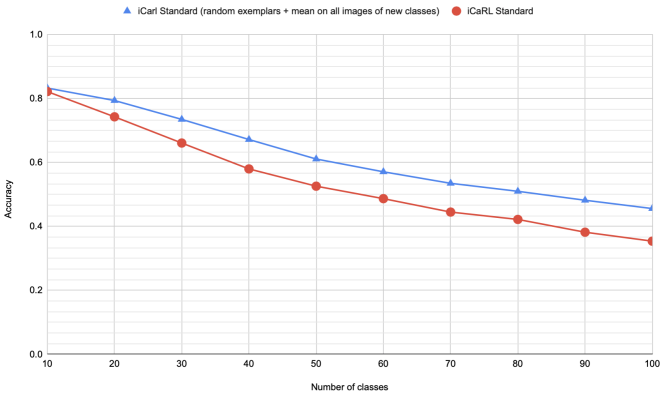

Figure 6. Comparison between iCaRL standard and iCaRL with our modifications

4. Our experiments on iCaRL

By strictly following the iCaRL paper we did not seem to be able to reach the results proposed: the overall trend was comparable but we were a few percentage points below in terms of accuracy, usually from the third batch onwards. For this reason, we decided to bring some changes to the standard implementation in order to try and reach the paper's results.

The first minor change we decided to make was relative to the computation of the class mean feature vector: instead of computing the class mean after having reduced and computed the new exemplar sets, we compute the class mean for new classes on all available training data when generating the exemplar set for that class and we compute the mean of existing exemplar sets before reducing them. This allows us to have a better representation of each class's mean by using more data without having to store more.

We also managed to achieve a considerable improvement (4%-5% in terms of accuracy from the second batch on) by selecting random exemplars: even though the iCaRL paper specifically states their herding method performs better than random sampling, this was in contrast with our experimental results.

We attributed this behaviour to the fact that random sampling should lead to more variance in a class representation, which could better describe the overall distribution of a class especially when considering only a few exemplars per set, as opposed to a more precise representation of the training data, but a less precise one when comparing the exemplars with our test data.

The strategies listed above were considered in place of iCaRL's approach for all our experiments and modifications.

4.1. iCaRL approach for the loss

The loss function used by iCaRL is a combination of two contributions: a **classification loss**, focused on the prediction for the new classes, and a **distillation loss**, focused on old classes scores obtained with the net at the previous step and which prevents us to lose all the information learnt previously. In order to account both these two contributions we first tried to follow the iCaRL's implementation, using **PyTorch's** BCEWithLogitsLoss: such loss implementation combines a sigmoid layer and the **BCELoss** in one single class.

$$l(x, y) = L = \{L_1, ..., L_T\}$$

$$l_n = -w_n \cdot [y_n \cdot log\sigma(x_n) + (1 - y_n) \cdot log(1 - \sigma(x_n))]$$

$$l(x, y) = \begin{cases} Mean(L) & \text{if reduction} = 'mean' \\ Sum(L) & \text{if reduction} = 'sum' \end{cases}$$

By using the **BCELoss**, each component's loss is indipendet with respect to the other componets[3]: this characteristic makes it a fitting choiche for multi-class classification problems. Such loss is called **B**inary **C**ross **E**ntropy **loss** due to the fact that, for every class, it sets up a binary classification problem and then it sums up the losses over all the different and independent binary problems. In our implementation we used as target of the **BCEWithLogitsLoss**, for each training image, the concatenation of the scores obtained on the old classes with the net at the previous step, passed through a Sigmoid layer to comply with PyTorch's implementation, and the one-hot encoded labels. By using this type of approach, instead of using two different losses for classification and distillation, the contribution of every class will be equally considered. As input for the BCELoss we use the scores of the fully connected layer of the net we are training, which is passed through a Sigmoid by the loss itself.

[3]the sigmoid layer flattens the vector in the range (0,1) and it is applied independently to each element of the vector, so each output vector component is indipendent from the others

4.2. Other approaches with the loss: Cross Entropy & KLDivLoss

As a first different approach, we tried using two different types of losses, the **Cross Entropy Loss** for classification and the **KLDivLoss** for distillation. The Cross Entropy loss measures the performances of a classification model, whose output is the probability of the element to belong to a class. The Cross Entropy Loss increases if the model produces a small probability value for the actual label or a large probability value for the wrong labels.

$$loss(x, class) = -log \cdot \left(\frac{\exp x[class]}{\sum_j \exp x[j]} \right) =$$

$$= -x[class] + log \left(\sum_j \exp x[j] \right)$$

Such approach penalizes predictions that are confident and wrong. We apply the **Cross Entropy loss** on both images of the new classes and exemplars of previous analyzed classes, providing as input the raw output of the net at the current state and as target the labels of the images. The **KLDivLoss** is a measure of how one probability distribution is different from another: this type of loss is a useful distance measure for continuous distribution. In our experiments we tried to apply this type of loss for the distillation component, comparing the output of the net at the previous step passed through a log_sigmoid or log_softmax with the scores on the old classes obtained with the net at the current state passed though a Sigmoid or Softmax[4] respectively, on the new data combined with exemplars.

$$l(x, y) = L = \{l_1, \dots, l_N\}$$

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

$$l(x, y) = \begin{cases} Mean(L) & \text{if reduction} = 'mean' \\ Sum(L) & \text{if reduction} = 'sum' \end{cases}$$

Using the Softmax function we noticed it was possible to achieve better results: such occurence may be due to the fact that we are considering a categorical multi-class classification problem.
Using the **BCELoss** we transform our problem into a binary classification problem, One-vs-All: for that case we use a Sigmoid because we do not care about our scores being a cumulative probability distribution.

[4]it is not necessary to apply log_sigmoid and log_softmax due to the fact that the **KLDivLoss** transform the probability values of the target automatically into log-probability values

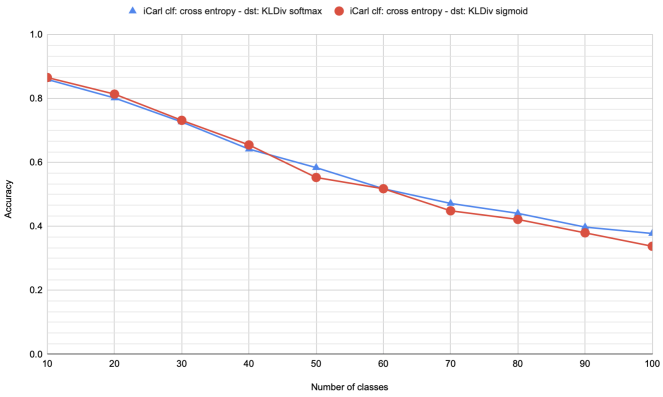

Figure 7. Comparison between clf loss: CE, dst: KLDiv both with Softamx and Sigmoid

4.3. Other approaches with the Loss: Cross Entropy & MSE

A different approach we considered was to use again the Cross Entropy Loss as classification combined with the **MSE** loss as distillation loss. The mean squared error loss is the average of the squared distances between our target variable and predicted values. The closer the loss is to zero, the better the prediction is. We considered two approaches with the MSE distillation:

- directly comparing the features of the images, extracted by the net at the previous step, with the features extracted at the current step during the update representation phase
- comparing the scores for each class, predicted by the net at the previous step and the net at the current step. Using KLDivLoss, we noticed that when comparing the scores it's better to pass them through a Softmax layer rather than through a Sigmoid, so we decided to do the same with MSE due their similar nature.

Observing our results, we notice that when using MSE on the features the accuracy decreases a lot analyzing the first batches of classes, but then in the end, on the last batches of classes, the accuracy is slightly better than using MSE on the scores. The results we achieve applying MSE to the scores are similar to the results achieved with KLDivLoss used as distillation.

4.4. Other approaches with the loss: MSE & MSE

As a last approach with the losses we tried applying the mean squared error loss as both classification and distillation loss. In order to use the MSE loss as a classification loss we provide as input to the MSE the scores predicted by the net at the current step and as target the one hot encoding labels of the images we were analyzing. As for the distillation loss, we compare the scores predicted by the net at

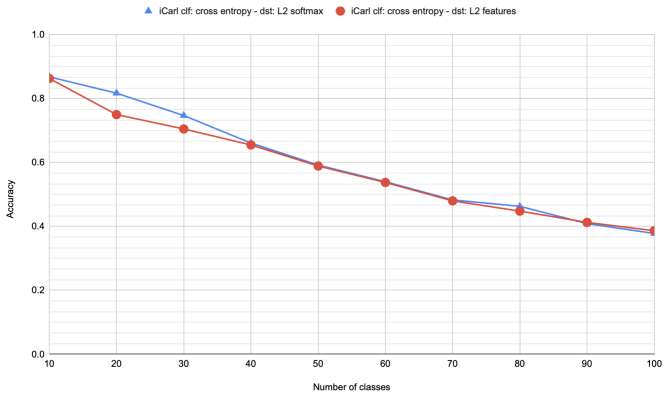

Figure 8. Comparison between clf loss: CE, dst: MSE both with Features and Softmax

previous step and the scores predicted by the net at the current step, passing both the scores through a Softmax layer.

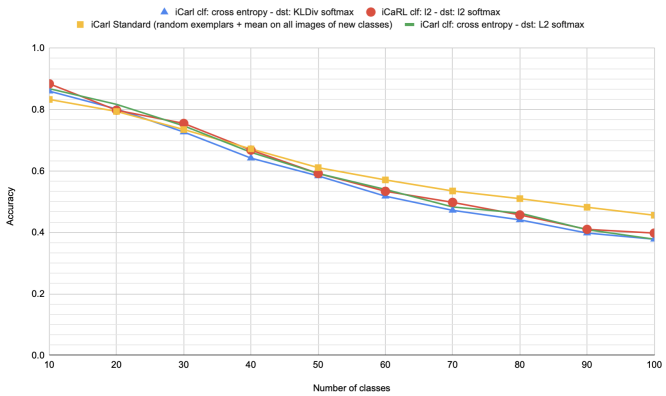

Figure 9. Comparison losses

4.5. Classifiers

After implementing iCaRL as reported in the original paper we proceeded by experimenting other classifying strategies:

- the first classifier we implemented was the **K-Nearest Neighbors**; KNN is a quite famous non-parametric classifier where class membership is decided by a majority voting by the K neighbors of the new point. This is done spatially representing data-points by means of their features.
  Due to the lack of such method in **PyTorch** we implemented it by computing the distance between new points and old data and then applying the class label of the majority of its $topK$ neighbors;

- we then proceeded by using a widely different and more complex classifier: the **MultiLayer Perceptron**. Such supervised algorithm is a feedforward neural network composed of multiple perceptron layers. Each node of each layer, except for the input layer, is a neuron that is triggered by an *activation function*. We used the methods given by torch to implement a vanilla MLP composed of three linear layers interspaced by RELUs. We decided to use such simple implementation for computational purposes and to have a higher degree of understanding of what was happening inside. We tried two different approaches: keeping the same number of nodes between the linear layers, and reducing the size of the hidden layer. Reducing the number of intermediate nodes we hoped to better filter out noise from the input features, but the difference between these two methods turned out to be negligible.

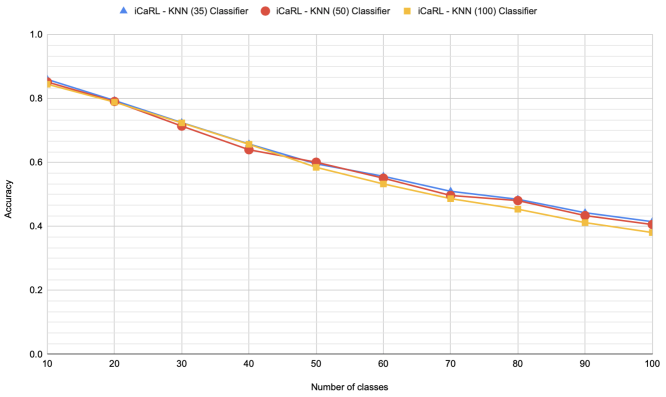

Figure 10. Accuracy of KNN classifiers, depending on K

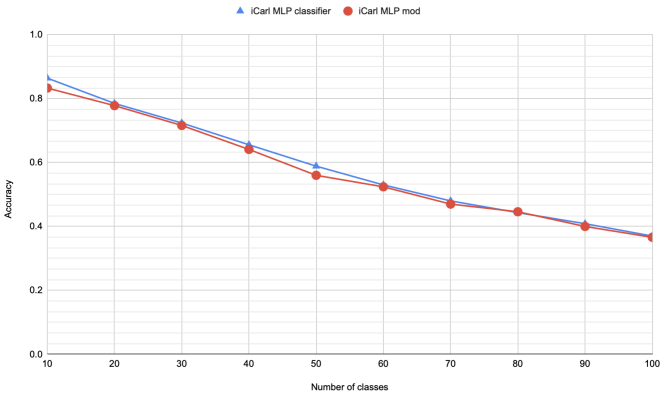

Figure 11. Accuracy of MLP classifiers

- finally, we modified the original implementation of the nearest exemplar mean classifier by using a cosine similarity as a measure. With such method we computed an estimate proportional to the cosine between the angles and used it to classify new data points with the nearest mean of exemplars approach proposed previously.

$$similarity = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

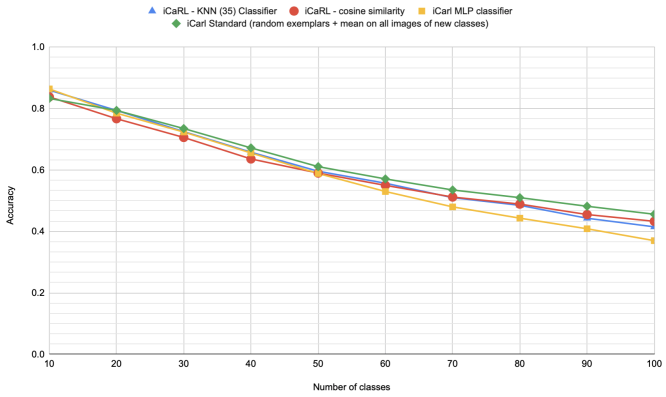

Figure 12. comparison between different classifiers

We see from the results in [*fig:12*] that the accuracy performance is quite similar between different methods and with the standard iCaRL implementation. The best performing classifier is the one using the cosine similarity: its affinity to the standard iCaRL implementation and the fact that magniture difference between features are of minor importance if compared to their direction. The second best performing classifier was the KNN with $K = 35$: such value appeared to be the one giving the best results while higher Ks probably brought to underfitting. As for the MultiLayer Perceptron, its worse performance may be caused by our data distribution, which does not allow the MLP to efficiently separate different classes.

4.6. Our proposals

Starting from the idea of better taking advantage of the exemplars, we tried experimenting with some ideas on how to preserve more features using the same number of exemplars or how to achieve similar results while using less. The general idea behind the following studies is to combine images at the lowest level possible, which is pixel by pixel, and then normalize according to the strategy we used.

4.6.1 Random images with same weight

We decided to do so by trying different strategies to generate synthetic data to obtain new, possibly more descriptive samples.
We first approached this strategy by trying to generate new synthetic data from random samples of each class and computing their mean.

Due to our lack of knowledge of what images were the most important and since the iCaRL implementation with the best results seemed to be the one using random exemplars, we decided to proceed without giving a higher weight to any particular image and sampling them completely at random. The degradation in accuracy was appalling, sometimes reaching even lower results than the Learning without

Algorithm 1 Random Sampling Average Image Generation
CONSTRUCTEXEMPLARSETMEANIMAGESAVERAGE

input images $X = \{x_1, \ldots, x_n\}$ of class y
input m target number of average images
input r number of images to average
  for $k = 1, \ldots, m$ **do**
    $Z = \{z_1, \ldots, z_r\}$ randomly sampled images from X
    $\mu_k \leftarrow \frac{1}{r} \sum_{i=1}^{r} z_i$
  end for
  $M \leftarrow (\mu_1, \ldots, \mu_m)$
output average image set M

Forgetting approach. After a checking the produced images we noticed that such procedure was extremely naive: some features could be observed, but the overall image was too noisy to be representative of the class.

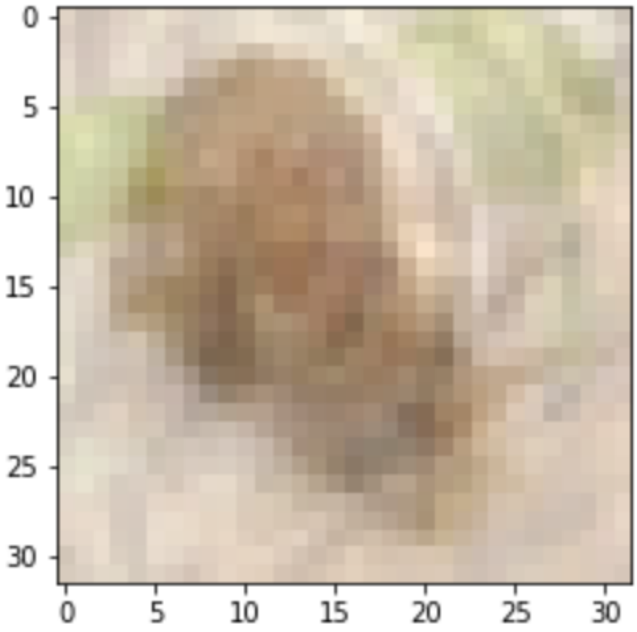

Figure 13. Sample image from class *Snail*

4.6.2 Clustered images with same weight

To find a way to condensate more information we consequently tried leveraging the similarities between images to create more meaningful data. At first many strategies for dataset distillation were analyzed: the main issue was to find a method to group images so that their combination wouldn't become an indiscernible mixture: to do so, given both our computational power and knowledge, we decided to leverage the neural network we used for classification to extract features from the datapoints. We then employed a simple clustering algorithm to divide them into

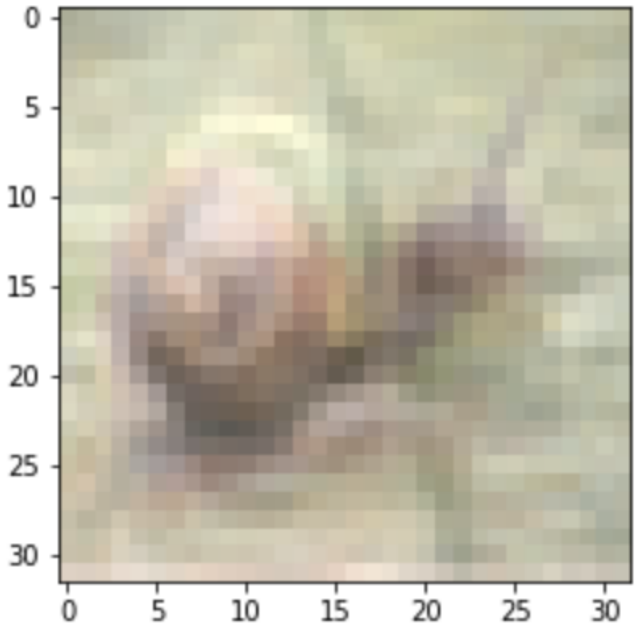

Figure 14. Sample image from class *Snail*

clusters based on their features distribution. By means of a **t-SNE**[1] dimensionality reduction, we plotted the distributions of both images and features of some sample classes, to get a general idea of the algorithm to use for clustering. Most of the classes turned out to be fairly uniform globular distributions when reduced with the **t-SNE**, hence we decided to make some experiments with K-means and infer what the best combination of parameters would be for our scenario.

We opted for the **K-means** clustering algorithm because we found it to be a good compromise between a fast and decent starting point given our brief analysis on different classes distributions.

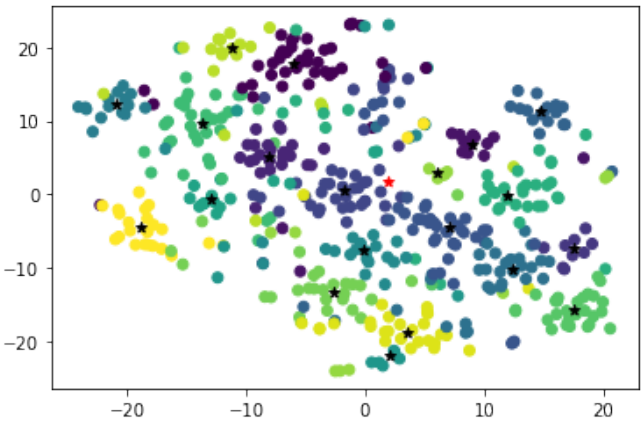

Figure 15. t-SNE representation of class 'pear'.
* represents class mean, * represent centroids

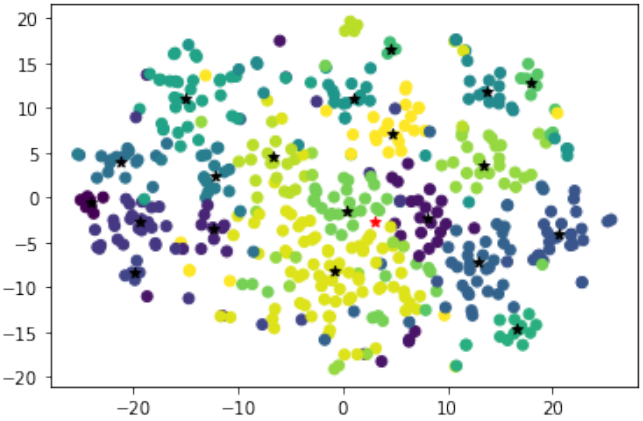

Figure 16. t-SNE representation of class 'racoon'.
* represents class mean, * represent centroids

Algorithm 2 Clustered images with same weight CONSTRUCTEXEMPLARSETMEANIMAGESAVERAGECLUSTER

input images $X = \{x_1, \ldots, x_n\}$ of class y
input m number of exemplars to generate
input l number of clusters
input f samples per cluster
  $C = \{c_1, \ldots, c_l\}$ clusters of images from X
  for $k = 1, \ldots, m$ **do**
    $Z \leftarrow$ collection of randomly sampled images from C clusters
    $\mu_k \leftarrow \frac{1}{\#Z} \sum_{i \in Z} z_i$
  end for
  $M \leftarrow (\mu_1, \ldots, \mu_m)$
output average image set M for class y

We started by clustering images based on their feature distribution with the idea to sample r random images from each cluster and compute their mean [*alg:2*]. By using this strategy we hoped to condensate more information in our synthetic data, but ended up creating excessively noisy images. Just like the previous approach, the overall image was poorly representative of its class.

At this point we started shifting from the idea of weighting different images uniformly to create a new image, as we could not seem to increase our accuracy score via this route.

4.6.3 Clustered images with distance-based weight

To try and solve this issue and give more importance to certain images, we opted to used a strategy where the sampled images we use to generate our synthetic data contribute to the image we are generating based on the distance of their cluster's centroid with respect to the class mean [*alg:3*]. By doing this, our goal was to value more the contribution of images whose features belong to a cluster closer to the class

mean, thus more representative of the distribution.
We tried three different strategies for sampling the images:

- from each cluster j, take k random images, compute their mean and add them to the image we are generating weighing them by a factor $\alpha_i = (centroid_i - class\_mean_i)^2$.
  By selecting some images from each cluster, we generate images with a contribution from each feature cluster, which should prevent loss of information when selecting exemplars.

- for each exemplar, select a random cluster j, take k random images from cluster j and add them to the image we are generating weighing them by a factor $\alpha_i = (features_i - centroid_i)^2$.
  By selecting images from the same cluster, we generate images as a linear combination of data-points with similar features, considering as most important the ones nearest to the centroid of the feature cluster.

- from each cluster j, take k random images, compute their contribution by weighing them by a factor $\alpha_i = (features_i - centroid_i)^2$
  and weight each cluster's contribution by weighing them by a factor $\beta_i = (centroid_i - class\_mean_i)^2$.

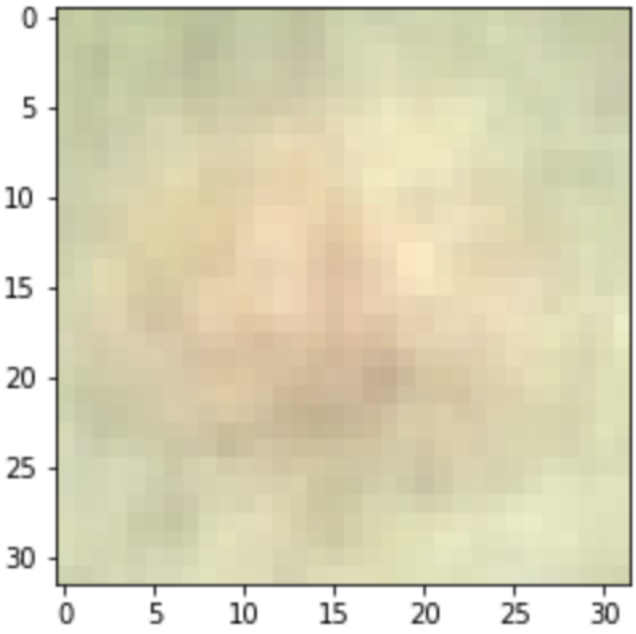

Figure 17. Sample image from class *Snail*

Neither of these methods performs as expected. In the first case we fall into the same issues we encountered in the previous approach, as we did not generate particularly meaningful images. All methods ended up producing monochrome images dominated by fuzzy noise. This method would probably perform much better with classes having images with very distinct features, but that's not our case.

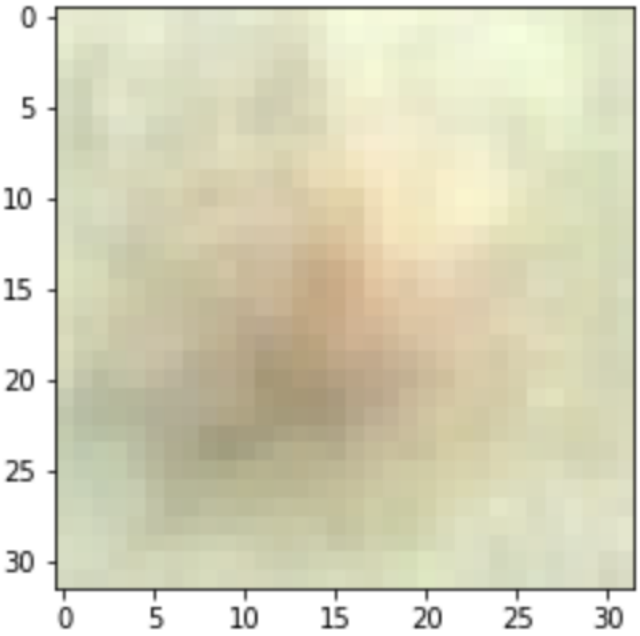

Figure 18. Sample image from class *Snail*

Algorithm 3 Clustered images with distance-based weight CONSTRUCTEXEMPLARSETMEANIMAGESCLUSTERSDISTANCE

input images $X = \{x_1, \ldots, x_n\}$ of class y
input m number of exemplars to generate
input r number of images to average per cluster
input l number of clusters
input f samples per cluster
  $C = \{c_1, \ldots, c_l\}$ clusters from X
  for $k = 1, \ldots, m$ **do**
    $Z_j \leftarrow$ collection of randomly sampled images from cluster C_j
  $\mu_k \leftarrow \frac{1}{\sum \alpha} \sum_{j \in C} \alpha_j \frac{1}{f} \sum_{i \in C_j} \mathrm{z}_j$ with α_i weight distance of the i-th datapoint
  end for
  $M \leftarrow (\mu_1, \ldots, \mu_m)$
output average image set M for class y

4.6.4 Random images with unbalanced weights

At last, we decided to proceed with a hybrid approach: as a matter of fact we needed to generate images that were diverse enough to be a decent summary of our class but that

contained a wide set of significant, and distinguishable, features. To do so we borrowed the idea from our best performing implementation of iCaRL and randomly sampled images from the entire distribution of our class; the first image is then weighted with a high coefficient, in our case 0.8, while the average of the other images would receive an overall weight of 0.2: those two were then summed and normalized. This strategy allows us to have samples whose features are clearly distinguishable while also adding additional information coming from random samples of the class [*alg:4*]. This strategy yields extremely interesting results: we obtain similar performances to our initial random exemplar sampling strategy, but when lowering the number of total exemplars K we notice an improvement with respect to the standard **iCaRL**'s strategy.
By using $K = 2000$ we maintain a comparable accuracy with respect to iCaRL, falling under by at most 2% on the last batches. This may be due to the fact that, when using a considerable amount of exemplars per class, we do not gain any advantage by using synthetic, noisier data and we end up losing some accuracy because of that. Lowering the total number of exemplars K on the other hand, allows us to take advantage of our more representative exemplars.
Already with $K = 1500$ we notice some appreciable differences between our method and iCaRL: when using less exemplars we suffer far less from the loss of additional information by compensating with more descriptive exemplars, as we can clearly see by analysing the differences shown in *fig:21*. By further experimenting on these scenarios, we noticed the previous statement holds: reducing to $K = 1000$ follows a similar behaviour as the previous case with 1500 exemplars, whereas by taking this concept to the extreme and considering only $K = 100$ exemplars better distinguishes between the two models, as seen in *fig:23*. When using so few exemplars we have a noticeable difference in performance starting from the first batches of 10 classes, and using the fully connected layer for classification instead of **nearest mean of exemplars** we notice further improvement, especially because having so few exemplars does not give an accurate enough representation when testing.

4.6.5 Clustered images with unbalanced weights

In order to generate a set of exemplars more representative of the distribution of our training dataset, we tried employing a clustering technique. In particular our algorithm, after having identified C clusters[5] for each exemplar, randomly selects one cluster and combine the nearest N image to the centroid [6]. Before the random selection of the N images

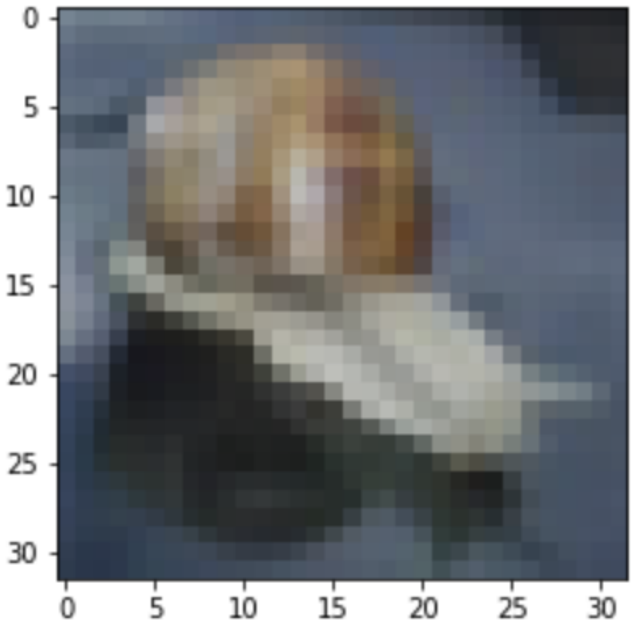

Figure 19. Sample image from class *Snail*

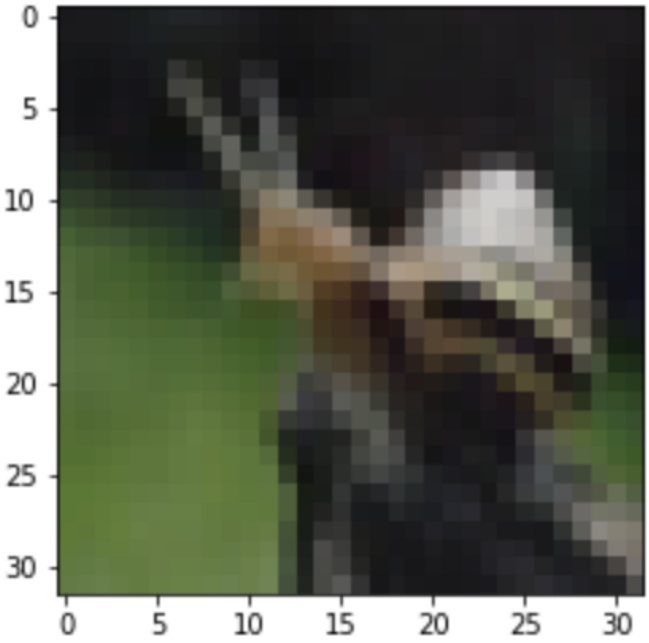

Figure 20. Sample image from class *Snail*

inside the cluster we purge from the list of images the nearest from the centroid: this is done in order not to to pick the same image as the nearest image to the centroid of the cluster, during the next iteration. The contribution of the centroid and the average of the other images is then combined in an unbalanced manner: the centroid will receive a weight of 0.8 while the average of all the others a weight of 0.2, or alternatively each one of the other images will

[5] similarly to previous strategies the clusters are computed in the feature space of the images of a single class

[6] the distances between the images and the centroids are computed in the feature space of the images of one class

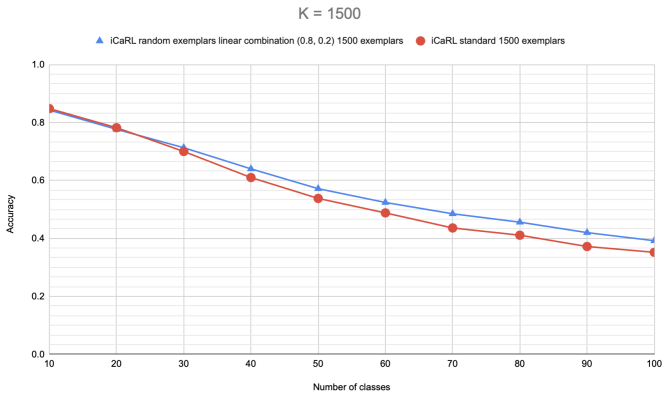

Figure 21. Accuracy of randomly sampled unbalanced exemplars vs iCaRL

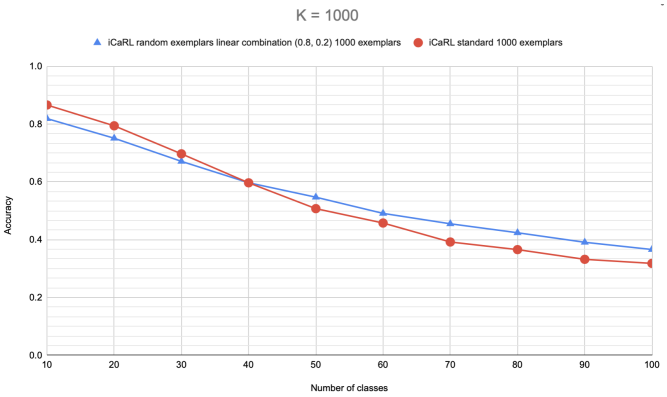

Figure 22. Accuracy of randomly sampled unbalanced exemplars vs iCaRL

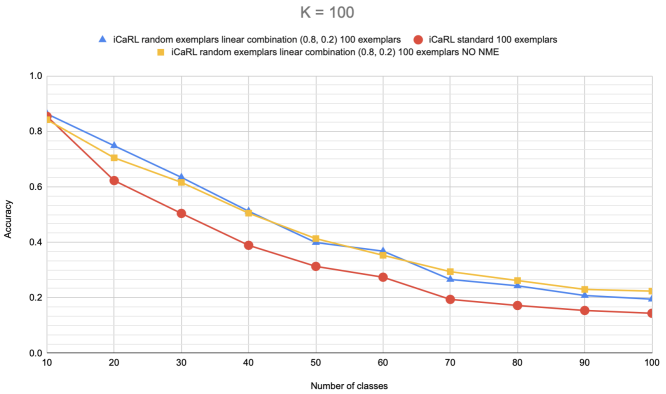

Figure 23. Accuracy of randomly sampled unbalanced exemplars vs iCaRL

receive a weight of $\frac{0.2}{N}$. Observing our results we notice that this technique does not provide an improvement in accuracy. Such curious occurrence probably happens due to the distribution of the data of the **CIFAR100** dataset: our clustering may not be that representative of the differences between the features in each class, thus selecting images by cluster does not translate in sampling the most significative

Algorithm 4 Random images with unbalanced weights CONSTRUCTEXEMPLARSETMEANIMAGESUNBALANCED

input images $X = \{x_1, \dots, x_n\}$ of class y
input m number of exemplars to generate
input s randomly samples per exemplar
  $Z \leftarrow X$
  for $k = 1, \dots, m$ **do**
    $C = \{c_1, \dots, c_s\}$ sampled images from Z
    $Z \leftarrow Z \setminus \{c_1\}$
    $\mu_k \leftarrow 0.8 \cdot c_1 + 0.2 \frac{1}{s-1} \cdot \sum_{i=2}^{s} c_i$
  end for
  $M \leftarrow (\mu_1, \dots, \mu_m)$
output average image set M

images of the class. Using this technique on a larger dataset with a different distribution might lead to better accuracy results.

We tried plotting our data by reducing its features with a **t-SNE** to try and gather more insight on the distribution of each class. Using a t-SNE allows us to better identify different clusters, while preserving local similarities. The t-SNE representation accentuates the distances between what we end up considering clusters, but in our case they seem to be relatively close to each other, which probably means all images of the same class have very similar features. This makes sense since we are considering rather few images of the same class (500 per class).

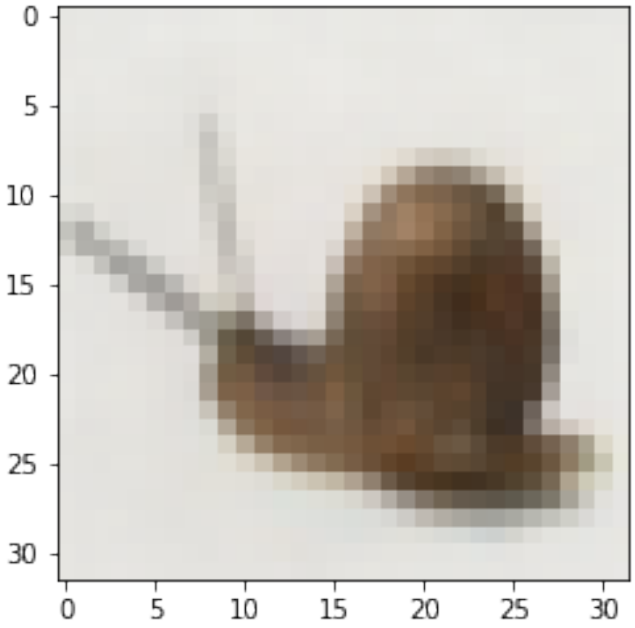

Figure 24. Sample image from class *Snail*

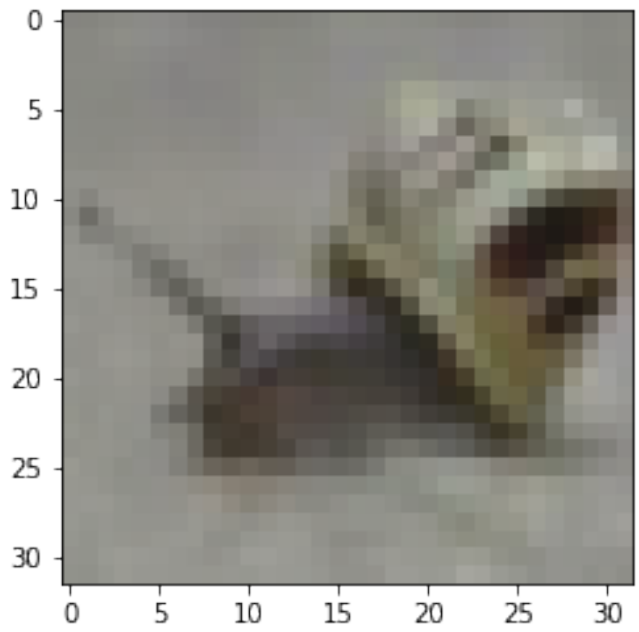

Figure 25. Sample image from class *Snail*

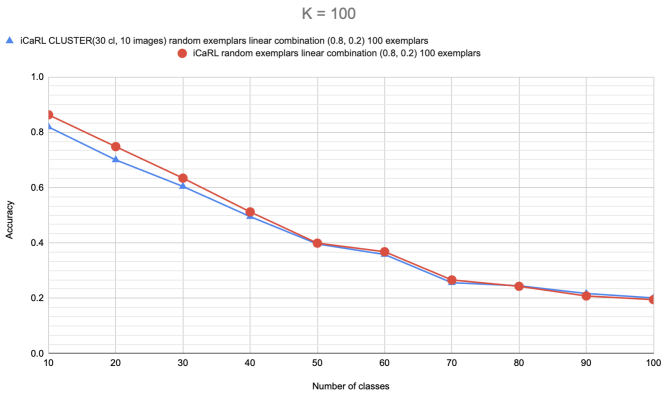

Figure 26. Accuracy of randomly sampled unbalanced exemplars vs cluster approach

4.6.6 Clustered images with unbalanced weights with separate feature extractor

To try and improve our results obtained with the clustering approach, we introduced a separate CNN of the same type we have used for iCaRL, which we train only with the batch of new classes, to obtain a better representation for new classes features without the influence of exemplars. This approach did not translate into an improvement in performance for the cases with very few exemplars ($e.g. K = 100$), as expected, and it produced negligible improvements for other cases, which does not justify the extreme time increment in the training phase.

Algorithm 5 Clustered images with unbalanced weights CONSTRUCTEXEMPLARSETMEANIMAGESCLUSTERSUNBALANCED

input images $X = \{x_1, \dots, x_n\}$ of class y
input m number of exemplars to generate
input r number of images to average per cluster
input l number of clusters
input f samples per cluster
  for $k = 1, \dots, l$ **do**
    $C = \{c_1, \dots, c_l\}$ clusters from X
  end for
  for $k = 1, \dots, m$ **do**
    $z_1 \leftarrow$ image closest to centroid of random cluster c_j
    $c_j \leftarrow \mathrm{c}_j \backslash \{\mathrm{z}_1\}$
    $\{z_2, \dots, z_f\} \leftarrow$ randomly sampled images from c_j
    $\mu_k \leftarrow 0.8 \cdot z_1 + 0.2 \frac{1}{f-1} \cdot \sum_{i=2}^{f} z_i$
  end for
  $M \leftarrow (\mu_1, \dots, \mu_m)$
output average image set M

5. Conclusions and possible future improvements

Our approaches for synthetic data generation produced some interesting results: even though we used some trivial techniques we were able to bring improvements in some scenarios, which leads us to think we could further increase the performances.
An interesting idea would be to employ *dataset distillation* [5] to try and generate even more representative synthetic data, so that we could further increase what we obtained by simply combining images on a pixel level.
Another fascinating strategy that could be worth analyzing is a generative approach by means of a Generative Adversarial Network [2], to create exemplars and avoid breaking the memory constraints we have on how many old samples we can keep.

References

[1] D. M. Chan, R. Rao, F. Huang, and J. F. Canny. t-sne-cuda: Gpu-accelerated t-sne and its applications to modern data. *CoRR*, abs/1807.11824, 2018.

[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1406.2661, June 2014.

[3] Z. Li and D. Hoiem. Learning without forgetting. *CoRR*, abs/1606.09282, 2016.

[4] S. Rebuffi, A. Kolesnikov, and C. H. Lampert. icarl: Incremental classifier and representation learning. *CoRR*, abs/1611.07725, 2016.

[5] B. Zhao, K. R. Mopuri, and H. Bilen. Dataset condensation with gradient matching, 2020.