Incremental Learning Project

Alessio Siciliano
Giulio Zabotto
Politecnico di Torino

Abstract

A major open problem on the road to artificial intelligence is the development of incrementally learning systems that learn about more and more concepts over time from a stream of data. In this work, we reproduce one of the most famous algorithms: iCaRL.

We show by experiments on CIFAR-100 data that iCaRL can learn many classes incrementally over a long period of time where other strategies quickly fail.

In the end we propose some changes to this algorithm that could increase the accuracy.

1. Incremental Learning Paradigm

Incremental learning is a method of machine learning in which input data is continuously used to extend the existing model's knowledge. It represents a dynamic technique of learning that can be applied when new classes become available gradually over time. The aim of incremental learning is for the learning model to adapt to new data without forgetting its existing knowledge. This approach to deep-learning tries to mimic reality, where it is demanded to continually learn new classes of objects.

This method faces *catastrophic forgetting*: assume a model has been properly trained on a given dataset; when retraining the model with a new set of classes with no permission to access the old dataset, the updated model is no longer able to recognize the old classes. When training the model with the new dataset, the current state of the model parameters is not considered and they are updated with regards only to new classes.

The *Incremental Learning* paradigm constraints follow the ones set by Rebuffi *et al.* in *iCaRL: Incremental Classifier and Representational Learning (iCaRL)* [5]:

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

2. Background

2.1. Distillation Loss

There were several attempts to overcome catastrophic forgetting. *Learning without Forgetting* [4] approaches this challenge applying the *Knowledge Distillation loss* [1] to the old classes. This kind of loss forces the model to approximate the output of another one. In our context, this loss can be used to mimic the output of the model trained on old classes, enabling the model to remember the old parameters.

LwF loss can be reformulated as the sum of the following two terms:

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

$$L_{dis}(x) = -\sum_{i=1}^{|C_0|} \tau_i(p^\star) log(\tau_i(p)) \tag{2}$$

In the first term, L_{ce}, C is the set of classes observed so far, y is the one-hot labels and p is the related probability yielded by a softmax function. In the second formula, C_0 is the set of old classes, $\tau_i(p^\star)$ is the softmax output of the old model for the old classes.

Being able to reproduce the output of the old model on the old classes prevents the network from over updating the parameters. Yet, the distillation loss is not perfect at reproducing the old output, hence some information is lost anyway. Moreover, it is shown that distillation loss is biased towards new classes since tends to classify test samples into new classes [5].

2.2. iCaRL

iCaRL model builds from the achievement of *LwF* and improves the results. *iCaRL* proposes a new classifier, and to add a small-sized memory to store images from old classes.

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

Nearest-Mean-of-Exemplars Classifier. *iCaRL* acknowledges that the classification process should be decoupled from the training. In such a way, the classification cannot be biased towards new or old classes. On the contrary, when classification is bounded to the training, it will be bounded to the updated weight parameters as well, that are related to the classes just learned.

The method proposed by *iCaRL* is the so-called *nearest-mean-of-exemplars* classifier. Employing Nearest Mean Exemplar classifier, *iCaRL* assumes that the meas is good estimator of similarity.

It computes the feature vector of the image that should be classified and assigns the class label with most similar prototype:

$$\hat{y} = \arg\min_{y} \|\phi(x) - \mu_y\| \tag{3}$$

where $\phi(\cdot)$ function is the feature extractor of a given neural network, μ for a given class y is the mean of its set of exemplars. For each set of exemplars S_y, $\mu_y = \frac{1}{|S_y|}\sum_{i=1}^{|S_y|}\phi(s_i)$, hence μ is the feature vector mean of the exemplars.

Also, to prevent comparing input samples with different weights, they are normalized.

How do we calculate the mean of each class? First, we compute the mean of the features extractor with the images stored in memory (without transformations) then we calculate the mean of the feature extractor again, but this time with the images flipped horizontally (*RandomHorizontalFlip*). Finally, we return the average of these two means.

Representational Learning. *ICaRL* makes use of exemplars during the training as well. First, it attaches the exemplar sets to the current dataset of new classes; second, it trains the model with a modified version of *LwF*'s loss. As in *LwF*, *iCaRL*'s loss is composed by a classification and a distillation loss. While *LwF* employs the cross-entropy loss, *iCaRL* decides to use the Binary Cross Entropy With Logits but we are in a multi-class setting so he has to use one hot encoding for labels.

$$\begin{aligned} L(\Theta) = -\sum_{x_i, y_i \in D} [&\sum_{y=z}^{t} \delta_{y=y_1} log(g_y(x_1)) + \delta_{y \neq y_i} log(1 - g_y(x_i)) \\ &+ \sum_{y=1}^{z-1} q_i^y log(g_y(x_1)) + (1 - q_i^y) log(1 - g_y(x_i))] \end{aligned} \tag{4}$$

Binary cross-entropy differs from the standard cross-entropy because it adds the output nodes that do not coincide with the sample's class, hence, it weighs such *negative nodes* as well. Considering every output node is a better choice when mimicking the output of the old model in the distillation loss; the more nodes it considers, the less room is left for updating the weights for the new classes.

Exemplar management in memory. The *iCaRL* model sets a memory to store samples of seen classes, thus preventing the neural network from excessively forgetting the old classes in the next batch. The choice of the exemplars follows from *iCaRL*'s classifiers. Since an input image is categorized into the nearest mean of the exemplars, one should select exemplars that most resemble the mean of the sample class. *iCaRL* asserts that the best approximation of the sample class distribution is achieved via *herding* [6]. Therefore, one for each seen class, exemplars are stored in the exemplar sets in a prioritized order: from the most similar sample to the mean of the distribution to the lesser, up until the memory allocated for that exemplar class is filled. When the number of seen classes increases, the memory for each class shrinks, since the memory for all the exemplars is fixed.

Random Exemplars Selection. We noticed that if we randomly choose exemplars, the accuracy decreases by about 1-2% which means that the *iCarl* selection algorithm works.

iCaRL model implementation. To set the baseline, we use the same neural network, the same hyperparameters, and dataset of *iCaRL*. The network is a 32-layers ResNet [2], exemplar memory size K is set to 2000, 70 epochs for each batch of classes, Stochastic Gradient Descent optimizer with starting learning rate of 2.0, then divided by 5 after the 49th and the 63rd epochs, and weight decay of 0.00001. The model is implemented with the PyTorch framework. The dataset used for experimenting is iCIFAR100 and we used the 10-classes benchmark protocol.

https://github.com/XxidroxX/Incremental-Learning

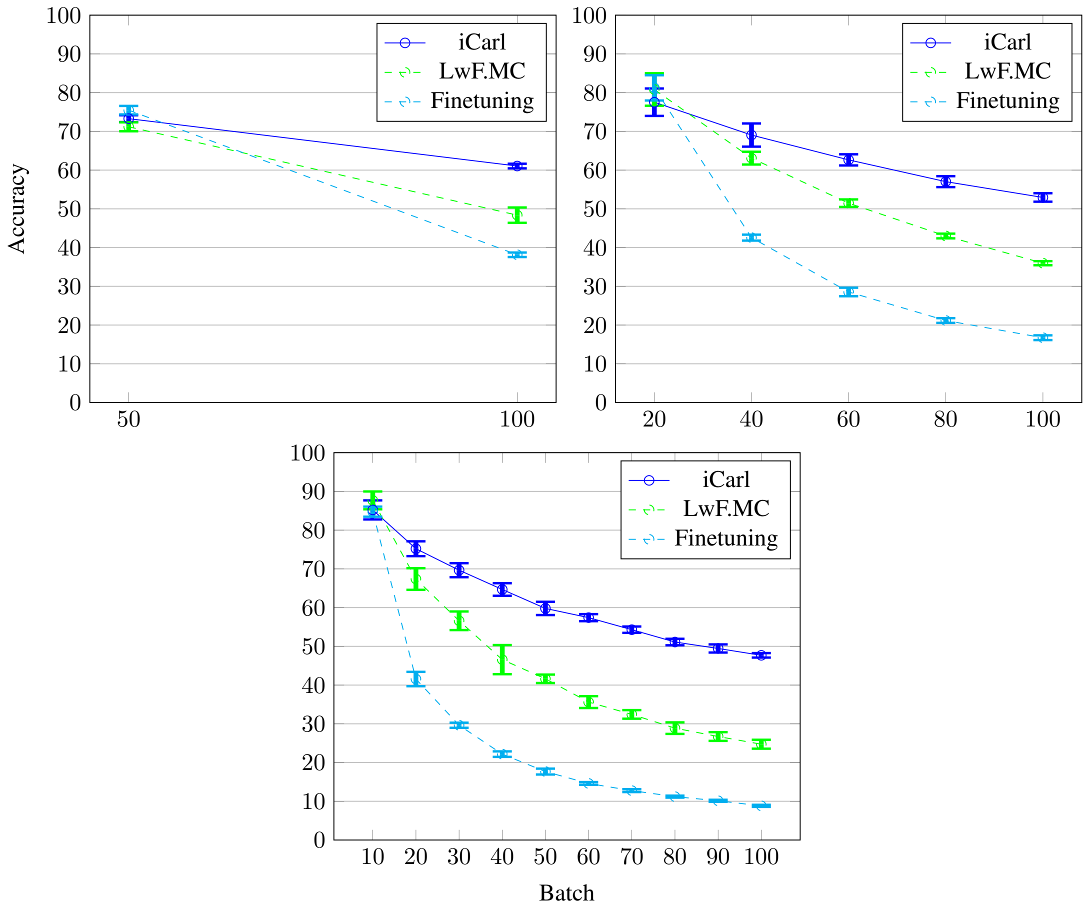

Figure 1. Experimental results of class-incremental training on iCIFAR-100. We successfully reproduced *iCaRL* experiments. *iCaRL* outperforms both *LwF-MC* and fine-tuning. None of the models achieves the *joint-training* accuracy of 68.6%, that is, training the model on the whole dataset at once.

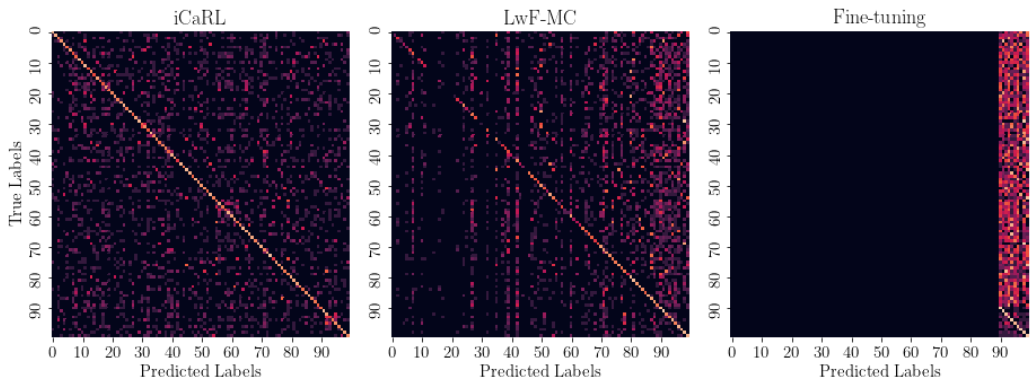

Figure 2. Confusion matrices yielded by different methods on iCIFAR-100 test dataset. To better visualize it, each entry has been transformed by $log(1 + x)$. As the uniformity of the patters in the *iCaRL* confusion matrix shows, it is not biased towards new classes, despite *LwF-MC*. On the other hand, fine-tuning clearly fails at the incremental learning task.

3. Ablation Studies

3.1. Evaluation Methods

Proper measurement discriminates well from bad experimenting. A standard metric for inferring a good-performing model is the *multi-class* accuracy. Nonetheless, in an incremental learning setting, it is not enough because matters which classes the neural network correctly classify, whether it identifies the old or the new classes. Confusion matrices help us discriminating against the old and the new ones. To distinguish a good model from a bad one, it is crucial to tell which classes the learner forgets from those which does not.

For each method, we ran the script five times and then we report only the mean for each batch of classes.

3.2. Classifiers

In this section we made some experiments changing only the classifier to show if *NCE* is really the best classifier or if there is another one that performs better and/or similar. For this task, we applied three different types of classifiers using only the exemplars in this way we can observe which method is better than the others.

Classify using only the exemplars. We tried to classify the new classes using all the images provided but this led to an imbalance dataset and the results were quite low. Hence, the solution was that to try with only the exemplars.

3.2.1 Support Vector Machine

The state of the art of classification for shallow learners is the support vector machine; high efficiency, high effectiveness. The underlying assumption is that dataset classes are separable up to a certain degree. For this classifier, we run the script with different values of the hyperparameter C that is the penalty parameter of the error term. It controls the trade off between smooth decision boundary and classifying the training points correctly.

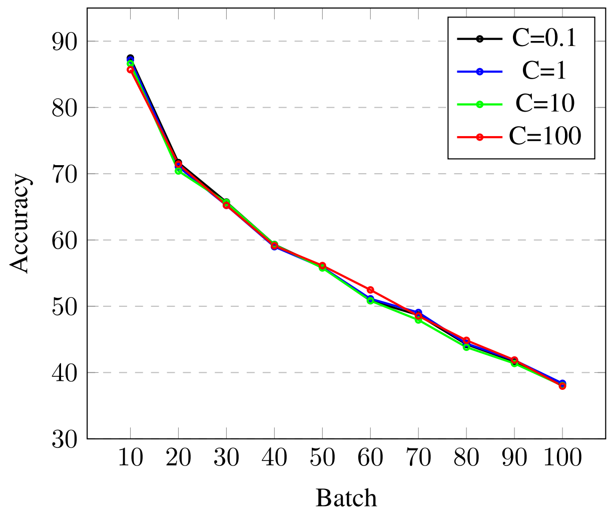

Figure 3. C hyperparameter equals to 1 yields the best results, yet it does not have a great impact on the performance

3.2.2 K-Nearest Neighbors

Since NME is based on the distance between a point and all the centroids of all classes known so far, an obvious choice is to use another famous classifier also based on distance: KNN. It is among the simplest of all machine learning algorithms: an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors and for this reason it is more complex and the choice of the value K will be very important.

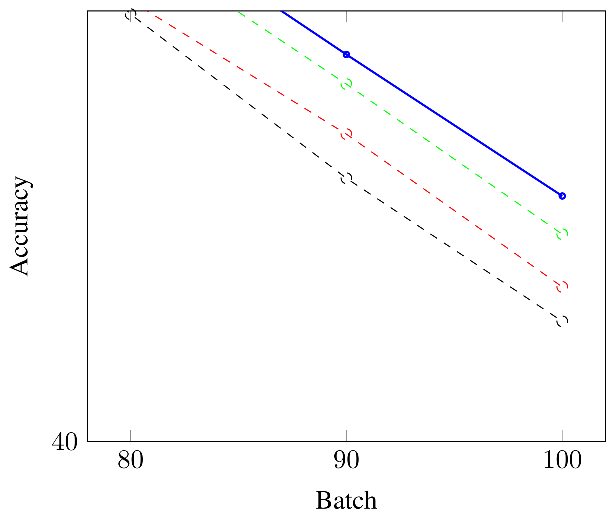

Figure 4. $K = 7$ ranks the highest in accuracy. Nonetheless, it does not yield such as improvement.

3.2.3 KNN + NCA

Since we are in a 64-dimensional space and KNN works better with a low-dimensional space we did a very interesting test to try to reduce the dimensions with NCA.

NCA. Neighborhood Component Analysis (NCA) is a machine learning algorithm for metric learning. It learns a linear transformation in a supervised fashion to improve the classification accuracy of a stochastic nearest neighbors rule in the transformed space.

Difference with PCA. The main difference is that PCA is an unsupervised algorithm whereas NCA is supervised (requires points with labels).

PCA and NCA, though both are linear transforms (that is, we end up with a D*d matrix which we can then multiply our data vectors with), optimize two different measures:

- PCA tries to preserve as much variance as possible.
- NCA finds the linear embedding where the KNN works best.

Unlike PCA, which is both convex and has an analytical solution, NCA is a non-convex optimization problem. As a consequence, every time we run NCA, we may get a different solution. As for K-Means and other non-convex algorithms, it is advisable to run it more than once and take the best solution. Unfortunately we didn't have time to take many measurements so NCA doesn't seem to improve KNN.

Combined with KNN, NCA is attractive for classification because it can naturally handle multi-class problems without any increase in the model size, and does not introduce additional parameters that require fine-tuning by the user.

NCA classification has been shown to work well in practice for datasets of varying size and difficulty. In contrast to related methods such as Linear Discriminant Analysis, NCA does not make any assumptions about the class distributions.

To use this model for classification, we combine a *NeighborhoodComponentsAnalysis* instance that learns the optimal transformation with a KNeighborsClassifier instance that performs the classification in the projected space.

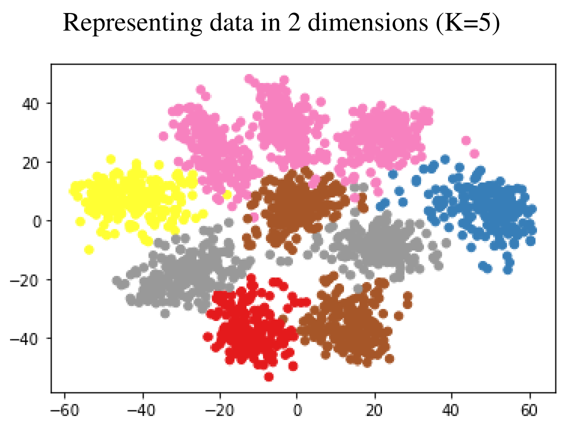

Figure 5. NCA seems to be useful to separate all the points from the first batch of classes as we can see in this figure.

If we set the dimensional space equal to 2, we get an accuracy on 78% on the first batch of classes and we can notice that we have a slight decrease caused by the drastic reduction in term of dimensions that removed too much information from the data. We can observe that when new classes arrive two dimensions are not more possible because there is an overlap between the old points and the new ones as we can see in the figure [6]. And this is demonstrated by the accuracy of the second batch that decreases to 43.45%.

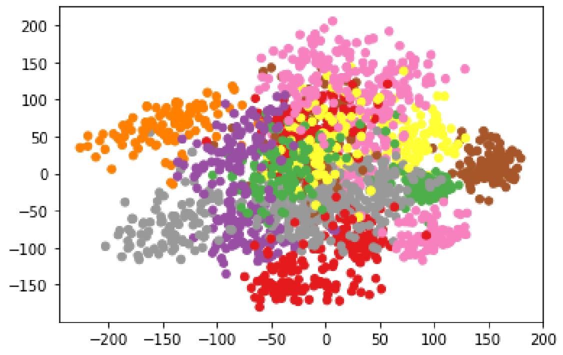

Figure 6. The more classes I have and the less they are separable.

Tests with different dimensions. We had no time to study better this method so we tried only with 64 dimensions and notice what changed.

	NCA		Standard KNN	
Classes	K=5	K=7	K=5	K=7
#10	86.10	87.60	87.40	87.10
#20	69.95	72.35	72.80	72.45
#30	66.63	67.70	68.13	68.47
#40	60.85	61.53	62.00	62.93
#50	56.20	56.62	60.20	59.50
#60	50.73	53.05	55.87	55.40
#70	54.26	53.95	53.57	53.60
#80	48.81	49.73	49.38	49.45
#90	45.13	46.43	46.81	47.10
#100	42.96	44.17	44.16	44.74

As we can see in the highlighted cells there is an apparently anomalous behavior of NCA where the accuracy instead of decreasing in the next batch increases, this is due to the fact that in that batch the points are better separable.

3.2.4 KNN and Cosine Similarity

We demonstrated that, with normalized data, applying KNN or Cosine Similarity is very similar. We can write the cosine similarity in this way:
$\frac{x^T * y}{\|x\| * \|y\|} = (\frac{x}{\|x\|})^T * \frac{y}{\|y\|}$ where the output is 1 if they are the same and goes to -1 if they are completely different, this definition is not technically a metric because it violates triangle inequality: $\|x + y\| \leq \|x\| + \|y\|$.

The euclidean distance can be equivalently written as $\sqrt{x^T * x + y^T * y - 2 * x^T * y}$. If we normalize every data point (as we did) before giving it to the KNeighborsClassifier, then $x^T * x = 1$ for all x.

Therefore, the euclidean distance will degrade to $\sqrt{2 - 2 * x^T * y}$. For exactly the same inputs, we would get $\sqrt{2 - 2 * 1} = 0$ and for complete opposites $\sqrt{2 - 2 * (-1)} = 2$. And it is clearly a simple shape, so you can get the same ordering as the cosine distance by normalizing our data and then using the euclidean distance. As long as we use the uniform weights option, the results will be identical to having used a correct Cosine Distance.

3.2.5 Multi-Layer Perceptron

An MLP consists of at least three layers of nodes - we will use three layers: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function (ReLU in this case). MLP utilizes a supervised learning technique called backpropagation for training.

Its multiple layers and non-linear activation distinguish MLP from a linear perceptron and it can distinguish data that is not linearly separable.

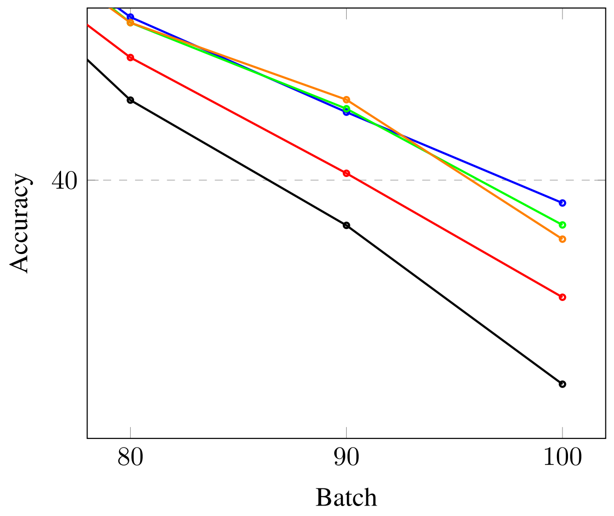

Figure 7. The best accuracy is achieved with 256 neurons in the hidden layers.

Final results:

	Classifiers			
Classes	**KNN**	**SVC**	**MLP**	**iCarl**
#10	87.10	87.29	84.63	85.20
#20	72.45	71.11	66.66	75.05
#30	68.47	65.22	64.28	69.72
#40	62.93	59.03	56.49	64.54
#50	59.50	55.99	54.23	59.88
#60	55.40	51.18	49.64	57.45
#70	53.60	49.14	47.55	54.31
#80	49.45	44.43	43.87	51.17
#90	47.10	41.95	41.61	49.58
#100	44.74	38.40	39.52	47.70

3.3. Loss functions

For all the experiments we used the PyTorch argument *reduction=mean* in each definition of the loss functions, in this way the losses are averaged across observations for each minibatch.

Cross Entropy Loss In a multi-class setting, the off-the-shelf loss for deep learning is cross entropy. To discover the difference of result with the *iCaRL*'s binary cross entropy, we tested it as classification loss. Despite being the general case of BinaryCrossEntropy, it slightly penalizes the results, though they are still comparable.

$$L_{ce}(x, class) = weight[class](-x[class] + log(\sum_j exp(x[j])) \tag{5}$$

Cross entropy offers to opportunity to weight the classes of the samples. Since classes are imbalanced due to the presence of the exemplars, we decided to weight each class with the inverse of its frequency. The results are still lower than the normal CE and the *iCaRL*'s binary cross entropy as we can see in figure [8]

Kullback-Leibler Divergence Loss KullBack-Leibler divergence measures the information difference between two statistical distributions. Despite being most used in statistics to determine the loss of information between two distributions, it can be used as a loss function in machine learning. In this setting, one assumes that the sample x and its label y belong to two different distributions. As for the Binary Cross Entropy, no output nodes are discarded.

$$L_{kldiv}(x, y) = \frac{1}{N} \sum_{i=1}^{N} y_i (log\ y_i - x_i) \tag{6}$$

We experimented with *KLDiv* for both classification and distillation loss. The results are slightly inferior to the Binary Cross Entropy used by *iCaRL*, yet still comparable.

Mean Squared Error Loss This criterion measures the mean squared error (squared L2 norm) between each element in the input x and target y.

$$L_{mse} = \frac{1}{N} \sum_{i=1}^{N} (y_i - x_i)^2 \tag{7}$$

We experimented with *MSE* only as classification loss, then as classification and distillation loss. The first experiment yielded the worst results among all the loss functions tried so far.

When outliers are present in a dataset, then the *MSE* loss does not perform well. Squaring the difference in length of an outlier will lead to a larger error, therefore it will not help discriminating well classes. In such a case, it is preferred to adopt the L1 loss function as it is less affected by the outliers or to remove the outliers and then use the L2 loss function.

Moreover, this loss derives from the regression of multivariate normal distribution, thus implicitly assuming that the underlying distribution of samples is normal. The bad results may be evidence that the sample distribution is not normal.

The second experiment results are enigmatic. The first batch of classes is as good as *iCaRL*'s binary cross entropy, while the second nearly halves the accuracy, and from that batch on it is nearly stable. This behavior may be due to the presence of many outliers in the second batch, but it does a persuading hypothesis.

Binary Cross Entropy Loss Binary Cross Entropy *BCE* is the loss function *iCaRL* employs. Although it is commonly used in binary problems, *iCaRL* applies it in a multi-class environment. In doing so, it not only considers the output yielded by the node whose label coincides with the ground truth of the input sample but all the other nodes as well. Having all the output of the old model, it may better mimic the old model output via the distillation loss.

Final results:

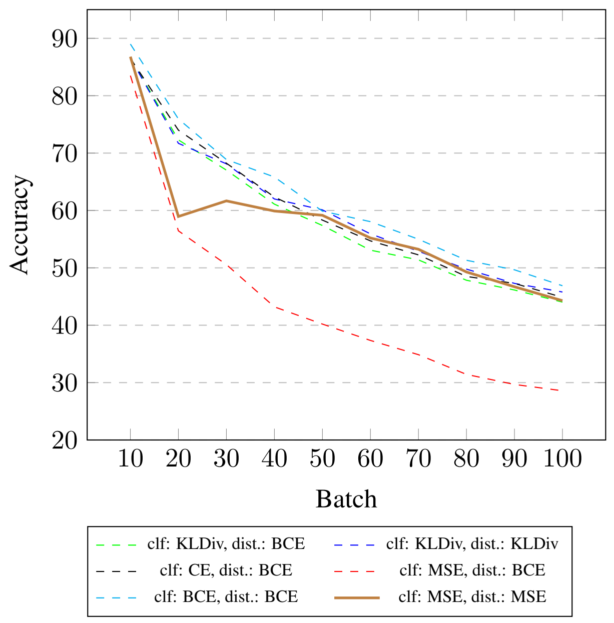

Figure 8. Accuracy on the training dataset when changing the loss function

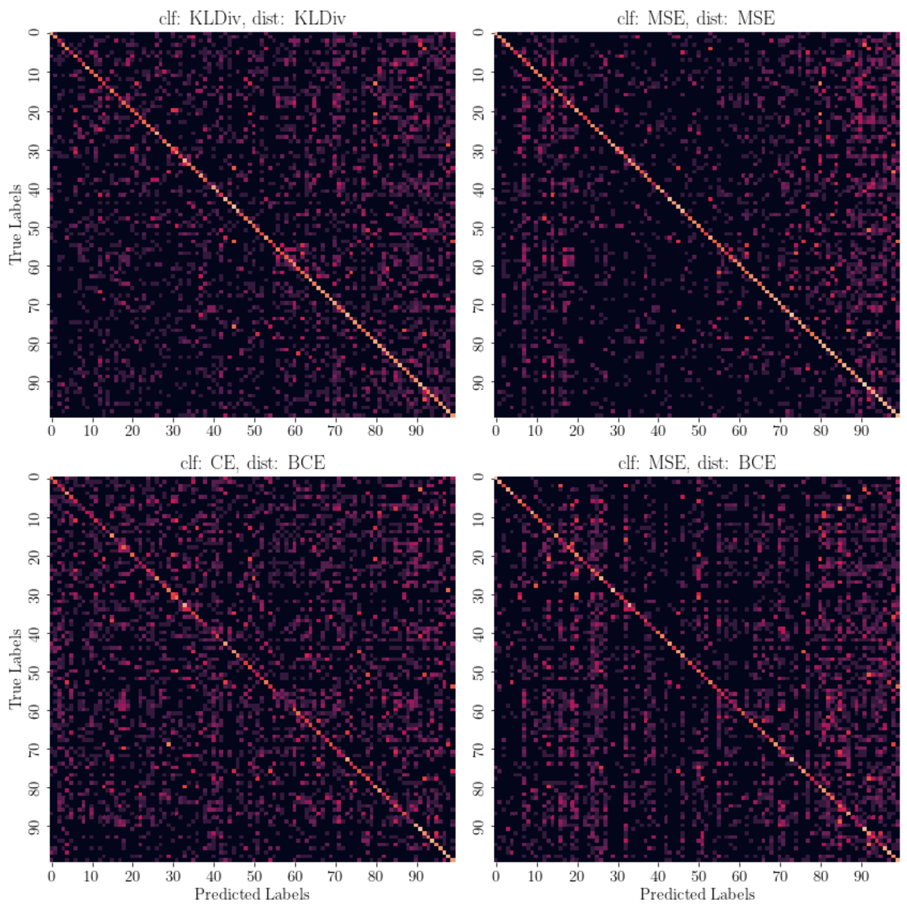

Figure 9. confusion matrix yielded by loss functions on the test dataset. To better visualize it, each entry has been transformed by $log(1+x)$.

4. Our Proposal

A key-point of *iCaRL* solution to incremental learning is the introduction of the exemplars. *iCaRL* uses a memory of K=2000 images for the exemplars, and at each step, it divides this memory equally among all old classes; batch by batch we have fewer exemplars per class while the new ones keep having 500 images and thus leading to a progressive imbalance of the classes during the training phase. To avoid or limit this phenomenon we adopted three different strategies:

- WeightedRandomSampler function.
- A huge data augmentation.
- Normalization of the weights of convolutional and FC layers.

4.1. Weight Random Sampler

Usually, when training a neural network, we process the images batch by batch for several epochs, so that we use all the images we have in the dataset once per epoch. In this case, having an imbalanced dataset, we use a sampler which allows us to assign a *weight* to each image, letting images with higher weights to be more likely extracted.

How to assign weights? This method will influence all the training phase. We must avoid adding more imbalance among the classes; as a consequence, the choice of the weights is very important.

We tried four different combinations of weights for the old classes and the new ones. For certain values, we have a small improvement compared to the accuracy of our implementation of *iCaRL*.

Here we will report all the combinations and the reasons:

- The simplest method we tried is to assign at each image a weight that corresponds to the dimension of the dataset (5000+2000) divided by 200 if that image is from an old class or 500 if it comes from a new class. This method yields the best result with a final accuracy of 47.60%.

- Weight based on the frequency of that class: we count how much a class appears into the dataset, then we assign to its images a weight equal to the inverse of its frequency. This approach is interesting, yet, batch by batch we notice why it is not useful. We recall that the number of images of each old class become less and less, so the frequency decreases from 200 to 22. As one can see, it leads to a high weight for old classes whereas the new classes have always 1/500 so we are in the opposite imbalance situations but this time towards the new classes.

- We tried to resolve the problem above by increasing, step by step, the weight of the new classes using the formula: $\frac{known\_classes/10}{500}$, while for the old classes we fix the weight to 1/200. Nonetheless, this approach decreases the accuracy to 45.72%.

- Finally, we tested a new combination assigning to the old classes a fixed weight of 1.1 and 0.9 for the new ones. Our intention was that to assign slightly more weights to the old classes than to the new ones in order to force the system to forget less the old classes. The accuracy scored 46.68% that is not so bad.

4.2. Data augmentation

Despite tackling the exemplars management as well, this approach differs from the previous because now we implement two standard data loaders, one for the exemplars and the other for the new images of the current dataset.

Why do we use two Dataloaders instead of one (as *iCaRL*) Behind this decision there is the our idea: we cannot save more images but we can scan multiple versions of the same image. In such a way, the parameters of the convolutional layers should be less susceptible to the class imbalances between the current new dataset and the exemplars. The exemplar and the current dataset differ in length, the former having 2000 images, the latter 5000. Because we wanted both datasets to have the same dimensions, we nested two loops, the first over the longer dataset, the second over the shorter dataset, that is, the exemplars. Once the outer loop ceases, so does the inner.

Because each exemplar image is loaded several times, we wanted to prevent the model from having exactly the same image, therefore, we apply a new image transformation among those already provided by *PyTorch*.

During the training phase with *RandomChoice* we choose and apply a transformation from the following list: *RandomCrop with padding*, *RandomHorizontalFlip*, *RandomGrayScale*, *RandomVerticalFlip*, *ColorJitter* and *RandomRotation*.

With all these transformations we decided to add more epochs to the training process so the system has enough time to process all versions of the images and can better generalize the content.

4.3. Normalization of weights of Convolutional/FC layers

The last attempt to tackle the class imbalance was an algebraic normalization of the weights. Ordinary ResNet applies the so-called batch-normalization, that is, it applies a z-score standardization, improving the overall performance of the model [3]. This statistical normalization does not prevent the model from having weights biased towards high-frequency classes. Our idea is that having weight arrays of unit length should correct this imbalance.

Moreover, normalizing the weights is recommended because it improves the conditioning of the optimization problem and speeds up the convergence of stochastic gradient descent.

To implement this normalization, we employ a function called *weight_norm* provided by *PyTorch* applied to each convolutional layer and to the only fully connected one.

The results are not satisfying since they are no better than *iCaRL*'s. We hypothesize that despite the unit weight of the parameter arrays, the relative weight of the components among the same array does not change, therefore the overall performance does not improve.

5. Conclusion

We introduced two different strategies for class-incremental learning: *LwF.MC* and *iCaRL*, where the latter gets the best results. Experiments on CIFAR-100 data show that *iCaRL* is able to learn incrementally over a long period of time where other methods fail quickly. The main reason for *iCaRL*'s strong classification results are its use of the exemplars. We were able to reproduce the same results as *iCaRL* and we proposed three different strategies to try to resolve some weakness we found in this algorithm.

In table [10], we show the accuracy with our methods compared to *iCaRL*. All these methods get more or less the same result but with the first approach we got a slightly higher accuracy. It is important to say that we tried to run the first two methods with the layers normalization but this did not lead to a significant improvement in term of final accuracy.

By doing all these tests we have a better understanding of *iCaRL* choices, and we believe that if one wants to improve iCarl performances its algorithm should be radically changed and simple changes are not enough.

	Our Proposals			
Classes	**Weight. R. S.**	**Data Aug.**	**Layers Norm.**	**iCarl**
#10	86.60	87.50	88.80	85.20
#20	75.60	71.65	72.15	75.05
#30	68.97	66.70	67.10	69.72
#40	63.05	60.80	62.45	64.54
#50	61.14	58.00	60.52	59.88
#60	58.30	56.15	55.82	57.45
#70	54.01	54.37	54.66	54.31
#80	51.03	51.30	51.59	51.17
#90	48.30	48.95	49.16	49.58
#100	47.60	46.89	47.44	47.70

Figure 10. In this table we reported only the mean of all the run we made. In some test, we noticed that with WeightR.S. we got the better accuracy.

References

[1] O. V. G.Hinton and J. Dean. Distilling the knowledge in a neural network. *NIPS Workshop*, 2014.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, abs/1502.03167, 2015.

[4] Z. Li and D. Hoiem. Learning without forgetting. *Computer Vision – ECCV 2016 Lecture Notes in Computer Science*, page 614–629, 2016.

[5] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. icarl: Incremental classifier and representation learning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[6] M. Welling. Herding dynamical weights to learn. *Proceedings of the 26th Annual International Conference on Machine Learning - ICML 09*, 2009.