Machine Learning and Deep Learning 2020
Incremental Learning

Sofia Borgato, Edoardo Pinna, Roberto Borgone

Abstract

We describe some incremental learning algorithm designed to incremental learn. We consider the particularly challenging form of this problem, where we assume that the previously generated data points are no longer available, even if some of those points may still be relevant in the new environment. Each algorithm employs a strategic weighting mechanism to update the weights of the CNN. We describe the implementation details of algorithms, and track its performance as accuracy functions. At the end we try to implement our own algorithm.

1. Introduction

Incremental Learning is a method in which the input data is continuously used to extend the existing model's knowledge. We train and then evaluate the performance of different algorithms on the *Cifar100* data-set of pytorch, that has 100 classes containing 600 images each.

1.1. Preliminary Steps

First of all we create a ad-hoc class for the Cifar100 data-set. For creating this new class we use the class Cifar100 provided by pytorch and modified the init method introducing a new parameter that permit us to access the instances of each label by the label index. We also add some method to the class as follow:

- __getitem__:that permit to access each images of the data-set by its index, and also convert the array representation of the image to a PIL image;
- len: return the len of the data-set(not the whole data-set but the portion you are considering in term of label index.

We also create two data-loader object (one for the train and one for the test set) and initialize resnet_cifar that is similar to the net described in the paper but adapt in pytorch (the net we referred was written in *theano*). We finally set the hyper-parameter that we are going to use during each experiments. We are using *Resnet32*, each Iteration consist of 70 epochs, the learning rate is 2.0 and is divided by 5 after 49 and 63 epoch, the batch size is set to 128 and weight_decay=0.00001.

2. Fine-tuning

As first try we implement a fine-tuning baseline for the incremental learning problem and we observed catastrophic forgetting.

2.1. Train and Test

We train the network on batches of 10 classes at a time, we have 100 classes so we retrained the model 10 times. Each time we evaluate on the test data corresponding to the classes seen so far (at first train we test on 10 classes, at the second train we test on the next 20 classes and so on). At the mean time we also calculate the accuracy and the loss for each epoch but this time evaluating only on the new classes added.

2.2. Results

As we expect we observe the catastrophic forgetting: we observe high accuracy for each epoch but the final accuracy calculated on the full set of the classes is really low. We also observe that the loss function calculated for each epoch converge to zero at the end of each train phase. This means that the network works good for the current classes set but for do this forgets whats learned in the previous train phase and so miss-classify the previous label. This phenomena is better visualized in Figure 1. In the first plot is represented the fast decrease of the accuracy testing on all the 100 classes. The accuracy starts form 74% and corresponds on the last 10 classes trained, when we finally arrive to the first 10 classes trained the net is unable to recognize them and the accuracy is nearby 10%.

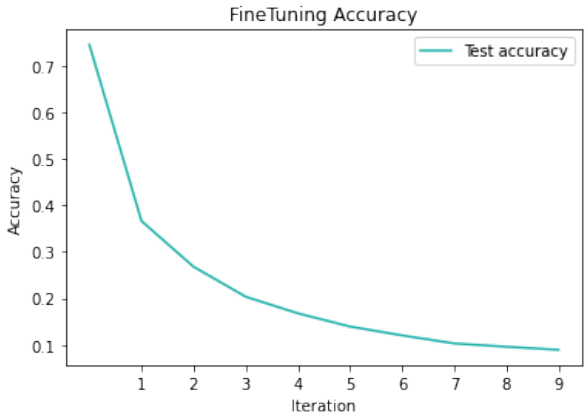
3. Learning Without Forgetting

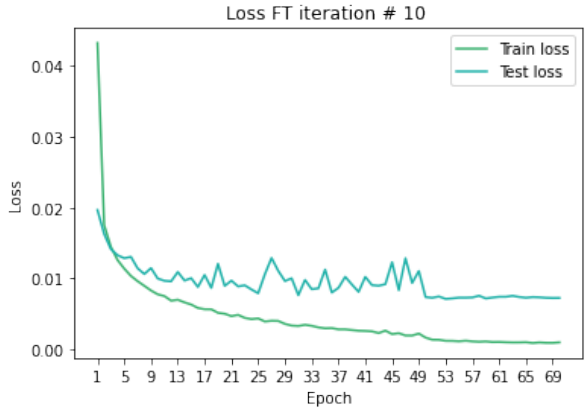
After reading the paper (Learning without Forgetting. Zhizhong Li, Derek Hoiem, Member, IEEE.) we understand that the main point of LwF is the implementation of loss function that permit to preserve the wight related to the class

trained before, this particular kind of loss is called distillation loss. In particular we use the following:

$$\mathscr{L}_{old}(y_o, \hat{y_o}) = -\sum_{i=1}^{l} y_o^{'(i)} \log \hat{y_o}^{'(i)} \tag{1}$$

where l is the number of label and $y_o^{'(i)}, \hat{y_o}^{'(i)}$ are the modified version of recorded and current probabilities. And





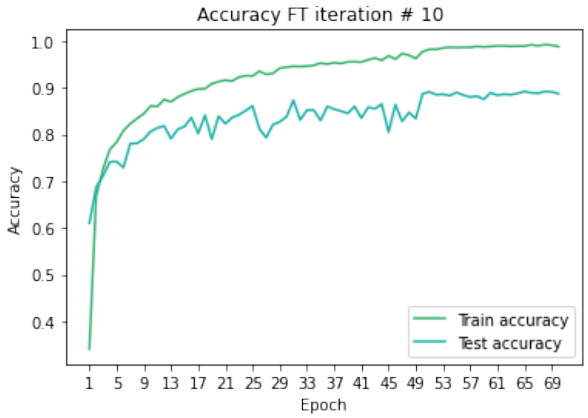


Figure 1. *FineTunning result*

$y_o^{(i)}, \hat{y_o}^{(i)}$:

$$y_o^{'(i)} = \frac{y_o^{(i)}}{\sum_j y_o^{(j)}} \tag{2}$$

$$\hat{y_o}^{'(i)} = \frac{\hat{y_o}^{(i)}}{\sum_j \hat{y_o}^{(j)}} \tag{3}$$

The total loss is defined as:

$$\lambda_o \mathscr{L}_{old}(Y_o, \hat{Y_o}) + \mathscr{L}_{new}(Y_n, \hat{Y_n}) \tag{4}$$

Where $\mathscr{L}_{new}(Y_n, \hat{Y_n})$ is a cross-entropy and λ_o is a loss balance weight.

3.1. Train

During the train phase we use an incremental learning strategy to train at each Iteration 10 new classes. First off all we initialize the *resnet32* with 10 classes (10 nodes in the fully connected layer). For each Iteration we train on the convolutional neuronal network for *num_*epoch computing the loss as explained thanks to the method *MultinomialLogisticLoss()* implemented by us exception with the first Iteration in which the loss is composed with only the losses referred to the current class (no previous knowledge to preserve). At the end of each Iteration we call an update classes function which increase the number of nodes in the fc-layer that permit us to train the new classes of the new data-set. Essentially the first iteration is a normal train on a CNN with 10 nodes in the fc-layer. After that we add 10 new nodes in the fc-layer and start a new Iteration. We forward pass the old network with the new data and we compute the losses with *MultinomialLogisticLoss()* method. Then we forward pass the new network with the new data and calculate the loss as a *CrossEntropyLoss()*. After summing the old and the new loss we are ready to backward the loss and compute the gradient.
With this approach the result was comparable with catastrophic forgetting. So we try to use the parameter lambda as in the equation (4) in order to balance the two different losses. After some attempts we choose as lambda 0.3.

3.2. Test

Before updating the number of classes and so the number of nodes of the fully connected layer we test the current state of our network using the the test method. The test set is composed by all the classes known by the network. We forward pass the network and at each epoch evaluate the accuracy and the loss function for each classes.

3.3. Result

In this first configuration we observed that by growing the value of lambda the net has difficult in training the new

data. This wearing is imputed to the net that try to remember to much the old classes and doesn't succeed in fitting the news ones. We can observe this behaviour in the plot in Figure:2 where the accuracy fast decay. We can also note how all the losses have the same behaviour.

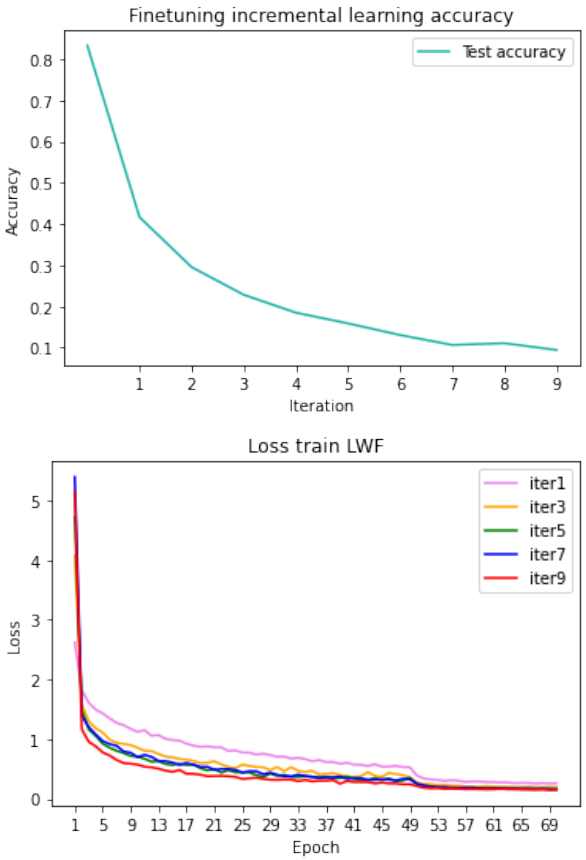


Figure 2. LWF with *MultinomialLogisticLoss()*

In order to solve this problem we try an other approach to compare the losses. This time for each epoch, for each batch we first forward pass old net with the new data and collect the output. We *torch.sigmoid()* this output with. Then we forward pass the new net (10 more nodes in the fully connected layer) and get the new outputs. Finally we compute the losses using as outputs the new outputs and as target the concatenation of the result of the sigmoid function with the *one-hot* representation of the new labels. Then we computes gradients. We register a good improve in terms of accuracy. The first plot in Figure:3 represent the accuracy of the final test for each 10 classes batch that decrease from the value of 85% to the values of 30%. This behavior is well explained with the second graph in which we are plotting the loss of the training set for different iterations. The loss of the first iteration converge to a smaller value with respect to the upcoming. This means that the net for taking into account the new classes is less effective in training the other.

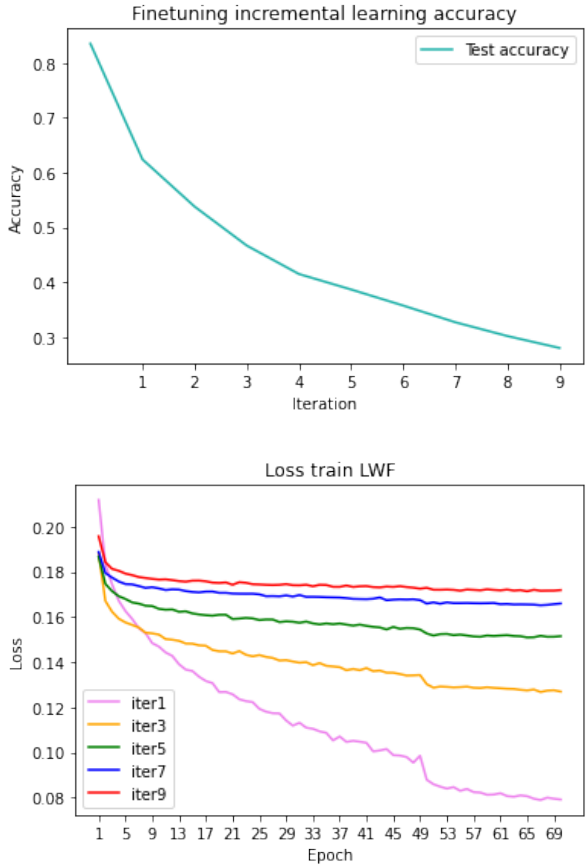


Figure 3. *LWF final model*

4. iCaRL

After reading the paper(iCaRL: Incremental Classifier and Representation Learning,rebuffi2016icarl) we understand that the implementation of iCaRL has some key points:

- Incremental Learning strategy: as described before we train on the first ten classes of the data-set, then at each Iteration we add tan new classes and train on the same network.
- The structure of the network change each time new classes are added: at each Iteration we add 10 nodes in the fc layer, as in LwF strategy. This step is necessary to avoid the forgetting of the old classes and we will better explain later.
- Exemplar set: the exemplar set is a set of images that is used at each Iteration in order to have a sort of sample of each old classes for the new train task and so at each iteration we train both the a new data-set composed by the exemplars set and the data for the new classes.

- The final classifier permit us to get the label of the input data. As we are going to see the network in iCaRL has just the function of features extractor.

4.1. Train

The whole training is composed by 10 different Iterations, at each Iteration we add and try to classify correctly 10 new classes. First of all we initialize the *ResNet32* with 10 nodes in the fc-layer. We train on the net the first 10 classes. We are using *BCEWithLogitsLoss* function and *SGDOptimizer* for each batch at the end of each epoch we normally update the loss function. At the end of the train phase we extract the exemplars set for each class. For to this we implement 2 different methods:

- random_exemplars: that choose randomly a defined number of images belonging to the same classes.
- herding_exemplars: we pass to this method the current status of the net and the images. For each label we forward pass the net and get the features vector. On all the feature vector of a classes we compute the class mean as

$$\mu = \frac{1}{m} \sum_{x \in X} \phi(x) \tag{5}$$

  Where x is the features representation of each images of the class X. Then we choose *m* images (we are going to discuss later in which way we choose m) one by one, minimizing the distances between the feature representation and the current class mean (each time we add an exemplar we also recalculated the current class mean), so we choose for all $k \leq m$ the *k-th* exemplar as :

$$\arg\min \left\| \mu - \frac{1}{k} \left[\phi(x) + \sum_{j=1}^{k-1} \phi(p_j) \right] \right\| \tag{6}$$

  where $\phi(x)$is the feature representation of the image x, and $\phi(p_j)$ are the features representations of the previous chosen exemplars. After choosing all the *m* features representation of the exemplars set we save the corresponding imagine in a list composed by all the exemplars sets of all the classes learnt.

After the exemplars extraction we are ready to start with 10 new classes in a new iteration phase. Before starting the training phase we concatenate the previous extracted exemplars set with the data of the new classes and we save the status of the previously trained net. We add 10 new nodes in the fc-layer initialized with random weights. For each epoch for each batch we forward pass the old network with the new data-set and we collect its outputs. Then we feed the new network with this same batch and collect the outputs. As last step we create a custom target vector composed for the first elements with the outputs of the old net, and for the last 10 with the one-hot representation of the labels of the new classes. Using this vector as target we can compute the losses and update weights. This trick is adopted because if we compute two different losses for classification and distillation loss the values of probability are unbalanced on the different number of classes we consider for each task.
The previous step ends the train on the network phase and we proceed to reduce the number of exemplars for each set in order to preserve memory. Given a fixed dimension of memory *G*, we want $G = m \cdot n$ (*n* is the number of classes currently known and *m* is the number of exemplars per classes) . Thanks to the method *reduce_* exemplars we remove the last exceeding exemplars for each classes and we are sure that this affects the less representative samples because the listed are ordered (when we are using herding strategy, in the random case it doesn't matter). With this new *m* values we can also call the method explained before to collect the exemplars set for the new classes.

4.2. Test

All the step discussed before are used for the feature extraction and to create the exemplars sets, for the classification task we implement *NME-classifier* (Near Mean of Exemplars). We pass to the classifier the last status of the network, the images we need to classify and the exemplars sets. Thanks to the net we get the feature vector representations (forward passing the network before the fc-layer: to do this we modified the structure of the forward method by adding a parameter, setting *parameter=True* we stop the forwarding before the fc-layer) of both the images to classify and of the exemplars sets. Then for each label we compute the mean of the feature representation. For each image we associate the label corresponding to the minimal distance to the class mean of each exemplars set. After all the Iteration we test on the test set composed by all the 100 classes the performance of the network. We use as hyper-parameters the set described in section 1.1. Just for the training set we adopt data augmentation applying *RandomCrop(32, padding=4)* and *RandomHorizontalFlip()* both with the default values of probability. For this attempt we use the random method for exemplars generation. In this configuration we get the result in Figure:4

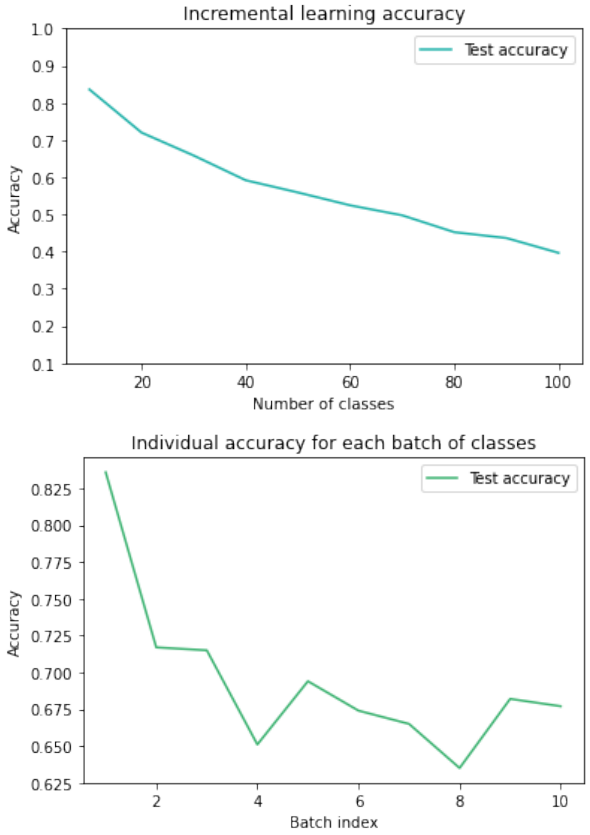


Figure 4. *iCaRL with random exemplars selection*

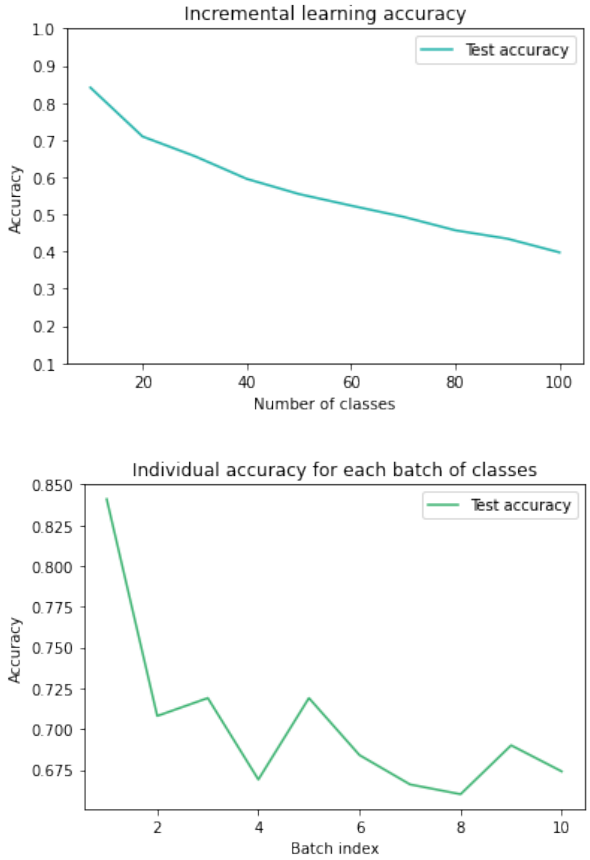


Figure 5. *iCaRL with Herding exemplars selection*

The plot represents the behaviour of the accuracy score during the test phase. To check the train status for each iteration we also print the values of the loss function for each 10 epochs and during each iteration it converge to zero. Finally with a test set composed with the new classes added at each Iteration we plot the flow of the accuracy score during the training (Figure4):

As we can see the accuracy for each batch of classes has a decrease. This is caused by the fact that the net is trying to remember the old classes and has some difficulties in train the new ones.
With all the same hyper-parameters we try with the herding method for the the generation of the exemplars sets. With the test described we get the plots in Figure 5.

As we can see there are no significant difference between the two different approaches. Even if it's just a small difference,by using the herding method the accuracy on all the 100 classes trained is higher and so we decided to maintain this method.

4.3. ICaRL with different set-up

With all the previous set-up we try to run iCaRL with different configuration. We try to change the classifier and the loss functions uses.

4.3.1 iCaRL with different classifier

First we try with a *KNN-Classifier* trained on the feature representation vectors of the exemplars sets. After few attempts we opt for the value of *K=5*, the result aren't comparable with those scored with *NME*. Probably this is caused by the feature representation of the exemplars belonging to different class are quiet "closed". And at the end of all the iteration we have just 20 exemplars each class. The result we get are represented in plot in Figure 6:

We also try to use *SVN* with *RBF-Kernel*. We tuned the hyper-parameter C and finally opt for the value 10. We use the SVMClassifier setting the kernel 'RBF'. The result are as bad as the *KNNClassifier* as expected because we are trying to find a way to separate 20 elements sets. When we test the final model the *SVMClassifier* try to separate vectors in a 100 dimensional space and to complete the task it has only 20 elements per classes that are obviously not enough. As the case before we plot the accuracy in Figure 7.

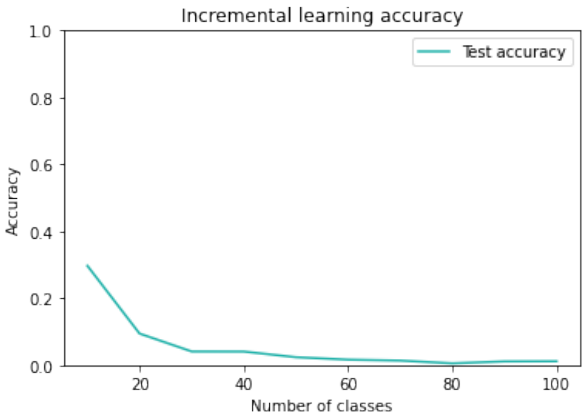


Figure 6. *iCaRL with KNN*

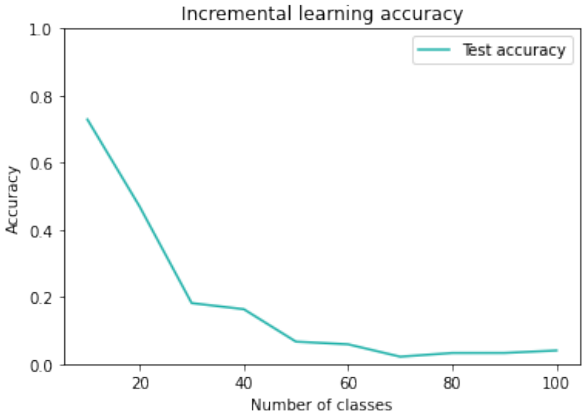


Figure 7. *iCaRL with SVM with RBF kernel*

4.3.2 iCaRL with different losses functions

In his implementation *iCaRL* uses *BCEWithLogitsLoss* that average the contribution of each classes. When we try with different losses we clearly understand the importance of this factor. We decide to maintain the structure described before in which we concatenate the vector with the old output with the one hot representation of the new output in order to create a custom target for computing loss. We experiment during *LWF* set-up that balance 2 different kind of loss is so difficult. For this reason we maintain this configuration in which we can use just one kind of loss function. First we try with the *L2Loss*, but after 2 iteration the accuracy is under the 40%. The *L2Loss* has a completely different structure compared with *BCEWithLogitsLoss*, for make it work well it's necessary a tuning of all the hyper-parameter and provide a way to balance the classes weight. We substantially modified the learning rate until the value of 1 and before back-warding we divide for the number of classes actually known. In this configuration we get the result in Figure 9

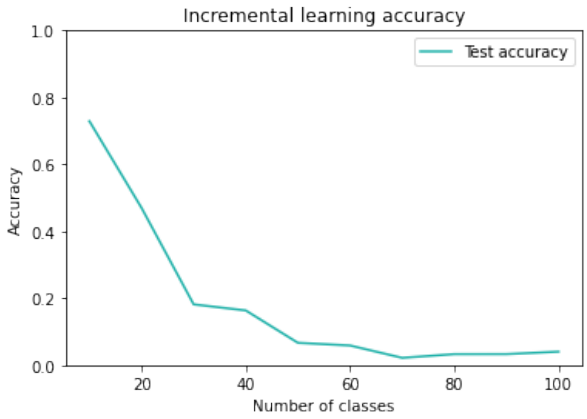


Figure 8. *iCaRL with L2Loss*

5. Our Approach-RES

From *Fine Tuning* we note how good is the network in classify 10 classes, and from *iCaRL* we appreciate his capacity in balance the weight of all the classes to avoid the catastrophic forgetting. For do that *iCaRL* modified the weight of the net. We decide to try to exploit the high performance of the net without modified the wight. We opt for initialize a new *Resnet32* each time a new data-set has to be trained (at each Iteration). Doing this the problem is shifted to the classify phase. We try to implement a classifier that works on the output vector of each net and choose one of the net. Finally we can make the prediction forward passing the net. The environment is similar to fine tuning pipeline except with at each Iteration we train on a new network. The hyper-parameter used are the same of the previous try.

5.1. How to choose the net

The main point is to define a discriminator between the outputs of the different net that can provide which net can better classify the image. As first try we compute the loss for each input images calculated by using as target the prediction made by each network and choose the minimum value. This approach completely fail. We try to build an index considering all the value in the output vector encourage the output with the max of its values more distant to all the other values. This because it's impossible to directly compare the outputs from different networks. Our index use all the values in the output vector, but each value is weight different. First we *torch.soft-max()*,we sort the output vector and then compute the index as follows:

$$index = o_o - \sum_{i=1}^{n} \frac{o_i}{i} \tag{7}$$

where i is the number of elements of the output vector, and o_i are the values. Now we compere the set of indexes for each image and taking the higher we choose the net. At

the end we use the chosen network to make the prediction. With just this management of the outputs the accuracy on all the classes decrease really fast. To better underline the difference between the first and the second element of the ordered output vector we also multiply the index for $\frac{o_1}{o_o}$. The result scored are better than *FineTuning*, but still not comparable with the two other algorithms. We think that an other possible solution is produce a sort of sample of the medium index calculated during the train phase of each net. And use this value to compute the best net to make the prediction of each images. Unfortunately we still haven't implement and evaluate it.

6. Compare different results

Finally we plot all the result scored with all the different method.

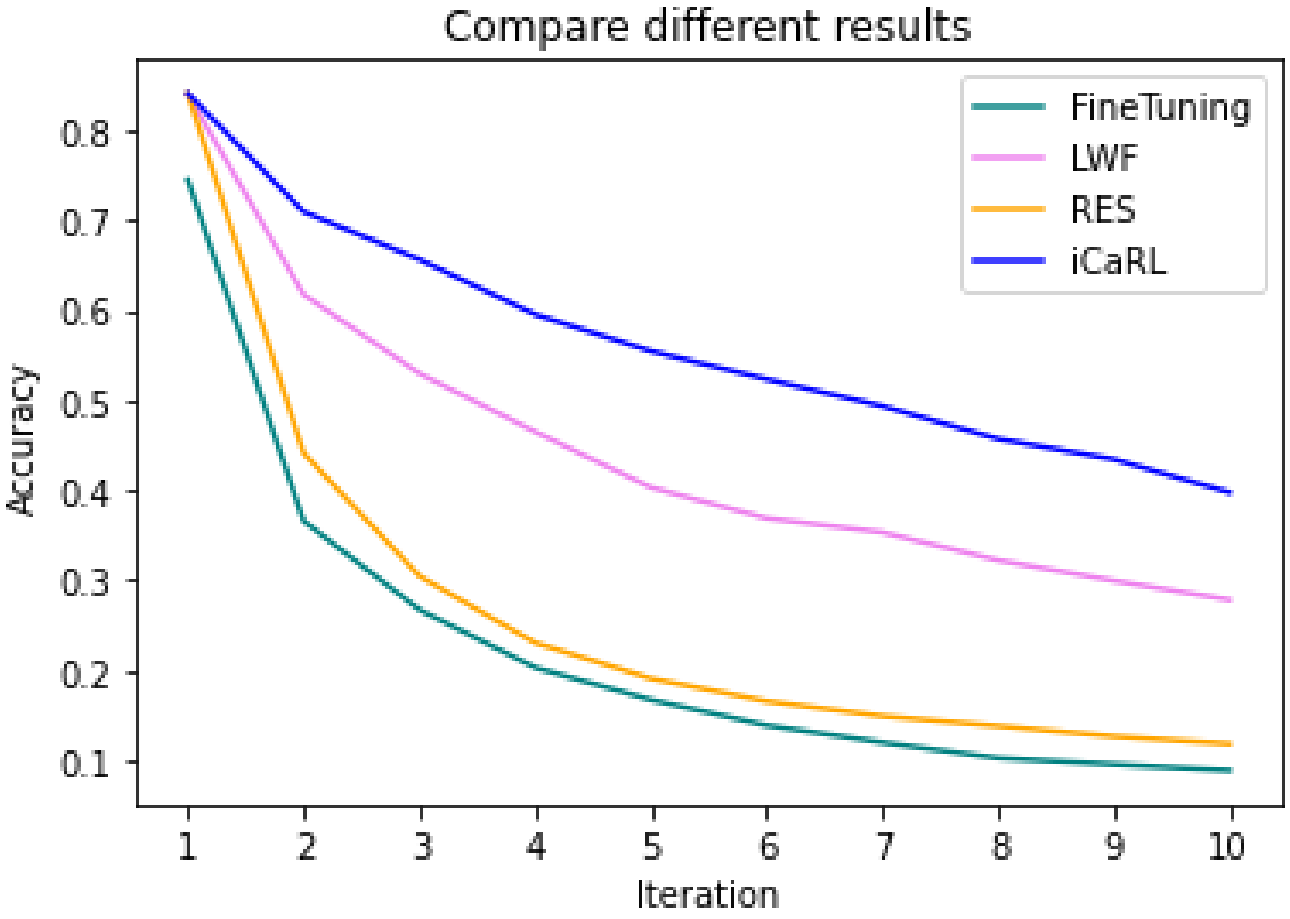


Figure 9. *Different Result*

7. References

@miscrebuffi2016icarl, title=iCaRL: Incremental Classifier and Representation Learning, author=Sylvestre-Alvise Rebuffi and Alexander Kolesnikov and Georg Sperl and Christoph H. Lampert, year=2016, eprint=1611.07725, archivePrefix=arXiv, primaryClass=cs.CV

@ARTICLE8107520, author=Z. Li and D. Hoiem, journal=IEEE Transactions on Pattern Analysis and Machine Intelligence, title=Learning without Forgetting, year=2018, volume=40, number=12, pages=2935-2947,

@articlepaszke2017automatic, title=Automatic differentiation in PyTorch, author=Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam, year=2017