POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Incremental Learning with Bayesian Neural Networks

Supervisors
prof. Elisa Ficarra
prof. Santa Di Cataldo
prof. Francesco Ponzio

Candidates
Valerio MIEULI
matricola 251372

Academic year 2019-2020

Ai miei genitori

Summary

In this period there is a lot of hype around the possibility to make computers able to learn automatic tasks by using a group of algorithms that go by the name of *Artificial Intelligence*. One of those tasks is the automatic classification of images and it is the one this work is focused on.
In real world applications, having an algorithm able to classify new types of images, once that it has been already trained for a previous task, would be extremely useful. Unfortunately, simply training the model with new types of images is not enough due to the so-called *catastrophic forgetting* problem: the algorithm will learn how to recognize the new types of images, but it will loose knowledge about previous ones.
Thinking about a real use-case affected by this problem is extremely easy: let us consider an oncologist who is using an application able to take histological images in input and to predict, among some already learnt types, the type of the cancer depicted in that image; at a certain point in time including the automatic recognition of new types of cancer becomes necessary, so the algorithm behind the application will be trained according this need.
The purpose of this work is to propose a possible solution to deal with the catastrophic forgetting problem and, in this way, to make a machine able to learn new images classification tasks *incrementally*.

Acknowledgements

Before diving into the seriousness of this work, I would like to take a moment to thank the professors who helped me in achieving a personally more than satisfying result. So thanks to profs. Elisa Ficarra, Santa Di Cataldo and Francesco Ponzio.

Contents

Chapter 1

Introduction

1.1 Incremental learning

The purpose of this work is to propose a possible solution to implement the incremental learning concept. In this case we are focused on a task of automatic classification of images. As we widely know now, for this kind of tasks neural network models are typically used and, in particular *convolutional neural networks*.

More specifically, we want to train a network in order to accomplish a given classification task first and, then, we want the train again the network because we want it to be able to execute a new classification task. This, of course, could be theoretically indefinitely extended, so we could ideally present to the network as many new classification tasks to be learnt as we want. The figure 1.1 represents the meaning of the just introduced *incremental learning* concept.

The first thing that one could think to deal with this *incremental learning*

problem is to train the current model as new tasks come as if we were dealing with a traditional learning problem. Unfortunately this trivial approach would fail because of the so-called *catastrophic forgetting* phenomenon. All neural networks suffer of this problem: when training a model for a task, but it has already been trained for a different one, the model starts learning about the newer one and, at the same time, it starts forgetting about the older one. The final result will be a good performance for the most recent task, but very bad one for the older task. Going very deeply in this explanation, when training the network for the new task, its weights start changing trying to maximize the performance according to the new classification problem and, for this reason, they could be moved even of a relatively huge quantity with respect to the initial values so that they could not represent the previous knowledge anymore.

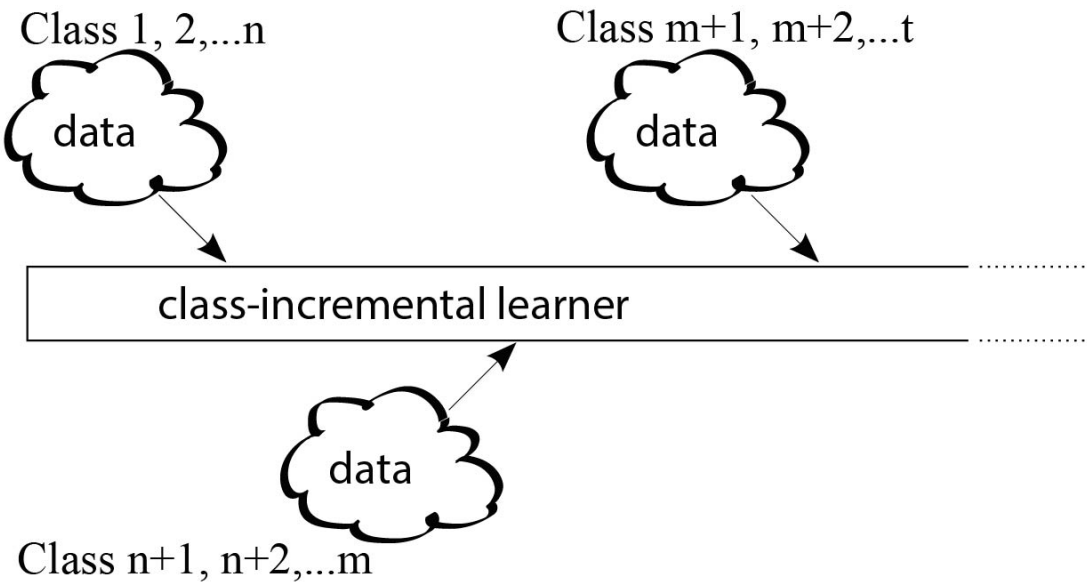

Figure 1.1. Graphical representation of incremental learning.

1.2 Formalization of the goal

At this point it should be quite clear in how many real-world applications an incremental learning approach would be extremely useful or even necessary. Now it is time to formalize which are the actual goals and issues of this topic. Basically we are looking for an algorithm having the following properties:

I. it should be trainable from a stream of data, properly divided in tasks, but with data of a single task that can occur in a non well defined order (within data of a task, samples are not necessarily presented to the model sorted by the type of class);

II. it should, at any time, provide a competitive multi-class classifier for the so far seen classes;

III. its computational requirements, specially the training time, and memory footprint should remain bounded or, at least, grow slowly as new classes are observed.

1.3 Trivial solutions

Once that we have formalized the goal of whatever would be a possible solution for the *incremental learning* problem, we can analyze it with respect of how much it complies with these aspects.
Here some trivial solutions are presented just to show how easy violating one of those aspects could be. For this reason, the following solutions can not be used in any real-world application actually.
The starting point is the same and it is a common convolutional neural network whose number of output nodes will increase as new classes are observed

in order to match exactly the number of different seen classes. We look at the network as split into two parts: a features extraction one which is followed by a classification part. The parameters of the features extraction part are shared among all tasks and identified by θ_s. Instead, for the classification part, the parameter of previous and new tasks are identified by θ_o and θ_n respectively. Said that, here we have the approaches:

- *Features Extraction* - θ_n are estimated by the training, while θ_s and θ_o are kept frozen. Typically performance on the new task are not so good since θ_s do not represent the knowledge about it, while the ones on old tasks do not change of course.

- *Fine Tuning* - θ_n are estimated by the training, θ_s are optimized by the training and θ_o are kept frozen. Typically performance on the new task are good since θ_s and θ_n have just been optimized and estimated respectively for it, but the ones on old tasks dramatically decrease since θ_s do not represent the knowledge about them anymore.

- *Joint Training* - All parameters are estimated from scratch every time a new task comes. Performance on both old and new classes could be good, but the solution is not always feasible. In fact for this approach all images of all old classes must be available, and so stored on the disk, and the training time quickly increase as new classes come.

Chapter 2

Available works

In literature we can find very many approaches concerning the *incremental learning* problem. Even if they can be even extremely different, they can be grouped in some quite well separated categories according to the main followed reasoning:

- additional loss terms to preserve the previous knowledge;
- a different sub-part of a network for each task;
- artificially generated samples of old classes to preserve the knowledge about them;
- expand the architecture for new tasks.

Here we are going to introduce some examples for each category in the next sections. At the end we will dedicate the entire next chapter to the last category in order to explain its aspects in detail since the implemented solutions belongs to that category.

Notice that, for each of the introduced solutions, the corresponding pseudo code and experiments' result are illustrated in the appendices of this work.

2.1 Additional loss terms

In this approach, typically there is a single network trained on a specific loss made by the traditional classification term and, in addition, a *distillation* one which helps the network to preserve the knowledge about previous tasks during the training for a new one.
Similarly to the trivial solutions, the network is considered as split into two parts: a features extractor followed by a classifier whose number of output nodes is the same of the number of all different seen classes.

2.1.1 Baseline - iCaRL

Incremental Classifier and Representation Learning [1] can be considered one of the milestones for incremental learning. It is characterized by three main elements:

- classification by a *nearest-mean-of-exemplars* rule;
- *prioritized exemplar selection*;
- representation learning using *knowledge distillation*.

The used architecture is the same of the one described in the previous section. In this approach some most representative samples of previous classes are stored up to the total number of them reaches the value of a parameter K. Setting this parameter the amount of the needed disk space is constrained

to a value that depends on the total amount of available space.

The first purpose of storing these samples is classification: at inference time the sample is forwarded through the network up to the features extraction layer, the mean vector for each class is computed and for the nearest one to the extracted features vector, the corresponding class will be considered as the predicted one.

The second purpose of storing these samples is training. It is done making the network to try to minimize a loss made of the traditional classification loss and by a *distillation loss*. The equation 2.1 represents the total loss and the second term, the distillation one, is computed using only the stored sample. They are forwarded through a copy of the network preceding the changes due to the current training phase then are used as ground truth for the principal network. The goal of this is to force the principal network to generate the same predictions on old samples both before and after the training for the new task which, in other words, means to reduce the *catastrophic forgetting* problem.

$$
\begin{aligned}
l(\Theta) = -\sum_{(x_i, y_i \in D)} [& \\
& \sum_{y=s}^{t} \delta_{y=y_i} \log g_y(x_i) + \delta_{y \neq y_i} \log g_y(1 - x_i) + \\
& \sum_{y=1}^{s-1} {q_i}^y \log g_y(x_i) + (1 - {q_i}^y) \log g_y(1 - x_i)]
\end{aligned}
\tag{2.1}
$$

A the end of a training session the set of stored samples is updated with the ones from the new classes and, for each class, only the most representative are kept. Of course reducing the number of stored samples per class could become necessary as new tasks are learnt in order not to violate the K constraint. How much a sample is representative for its class is computed by

the distance of its extracted features vector from the mean features vector of that class.

Talking about overhead, the most part of the one of this solution is about computation of most representative samples at the end of the training procedure because this must be done for all seen classes. Of course we have a part of it even during training since the training time increase with the number of tasks, but typically new samples are much more then the ones store per each one of the previous classes.

Another important aspect to consider is the K parameter. Both the performances and the overhead depend on it: the more is K, the more is the knowledge preserved about old tasks but also the more is the overhead and the needed disk space (and vice versa). For these reasons, when implementing a solution based on *iCaRL*, how fast the training phase needs to be and the tolerable percentage of errors of the algorithm should be taken into account according to which are the available resources.

One of the main problems about *iCaRL* is that typically, regardless the value of K, the training dataset is unbalanced towards the task samples. As we know, this could lead the network's predictions to be more oriented towards the new classes during the training step. This, of course, leads to poor performances at inference time considering all seen classes.

In the next sections two possible improvements are presented and it is important to remember that in both of them a limited amount of old samples is stored.

2.1.2 End-to-End Incremental Learning

The *End-to-End Incremental Learning* [2] training procedure simply has an additional fine-tuning step with respect the *iCaRL*'s one. In fact, after the training with all new and a small selection of stored old samples, a reduced and balanced dataset with both types of samples is built. The just created dataset is then used to fine-tune the model. The last step of the training procedure is updating the representative samples for all classes seen so far. Also for this step there is a little difference with respect to *iCaRL*: in this case the metric measuring how much a sample is representative is the prediction score of the network for that sample.

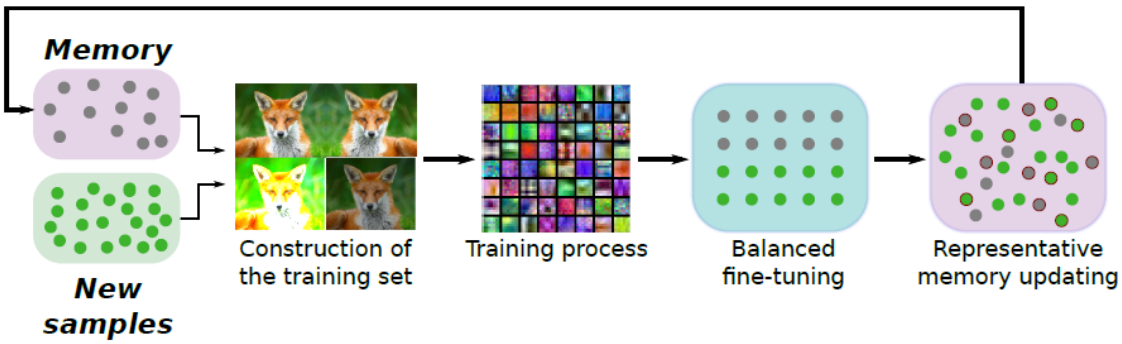

Figure 2.1. The whole training procedure of End-to-End Incremental Learning.

2.1.3 Large Scale Incremental Learning

In the approached followed by *Large Scale Incremental Learning* [3] before the last fully connected layer a bias correction one is introduced and it is simply a linear model with only two parameters to estimate. From the whole training set a much smaller validation set is taken out and it is balanced set. The idea is that the balanced validation set should quite well approximate

the distribution of old and new classes for real data. The training procedure is split into two stages:

1. train, with both classification and distillation loss, the whole network while keeping the bias correction layer frozen by using the remaining training data;

2. train the bias correction layer while keeping all the network but such a layer frozen by using the small validation set.

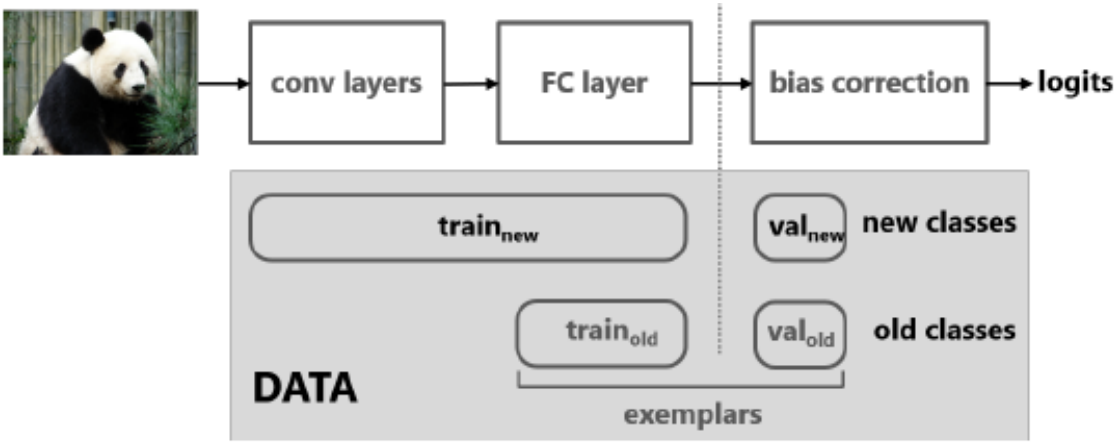

Figure 2.2. The two steps training procedure of Large Scale Incremental Learning.

The linear model that implements the bias correction is the following one (o_k is the output logits for old classes) and it is trained using the loss function in the equation 2.3:

$$q_k = \begin{cases} o_k, & 1 \leq k \leq n \text{ (old classes)} \\ \alpha o_k + \beta, & n+1 \leq k \leq n+m \text{ (new classes)} \end{cases} \tag{2.2}$$

$$L_b = -\sum_{k=1}^{n+m} \delta_{y=k} \log[softmax(q_K)] \tag{2.3}$$

As last thing, notice that the bias correction layer has thought with only two parameters due to the little size of the validation set. In fact, considering a bigger validation set would lead to a smaller training set and so to a worse result on the first step of the training procedure. At the same time, if the bias correction layer had more than two parameters, it would be more complex to well estimate them with such a few validation samples.

2.2 A different sub-part of the network for each task

Here, fixed the initial network, for each new task, a different part of that network is used. More specifically a different group of weights of the same network is used for each task.

In this work, two examples of this approach are going to be examined which differ for a specific aspect: weights could or could not be shared for different tasks. Despite this important difference, these methods have common positive and negative aspects: the size of the initial model does not change significantly as new tasks come but, at the same time, the model could not learn new tasks indefinitely due to saturation of its capacity; performances on old tasks are not changed by adding new ones; no samples from old tasks are stored.

In general these methods are the one requiring least disk space and training time since at each training procedure only data of the current task are involved.

2.2.1 Packnet

In this method [4], for each new task to be learnt, the training procedure works as follows:

1. train the network for the new task by using all remaining weights;

2. prune some of the just trained weights (this leads to a decreasing in accuracy for the current task);

3. train the network again using the remaining parameters (in order to let the on the current task accuracy to increase again).

Going more deeply in the pruning operation, once the percentage of weights to be cut for each layer has been selected, they are pruned starting form the ones with the lowest absolute magnitude which are less relevant for the final output. At the end of this process a matrix per task is stored in order to indicate which weights should be considered at inference time for that task. This is a very little overhead in both time and resources terms.

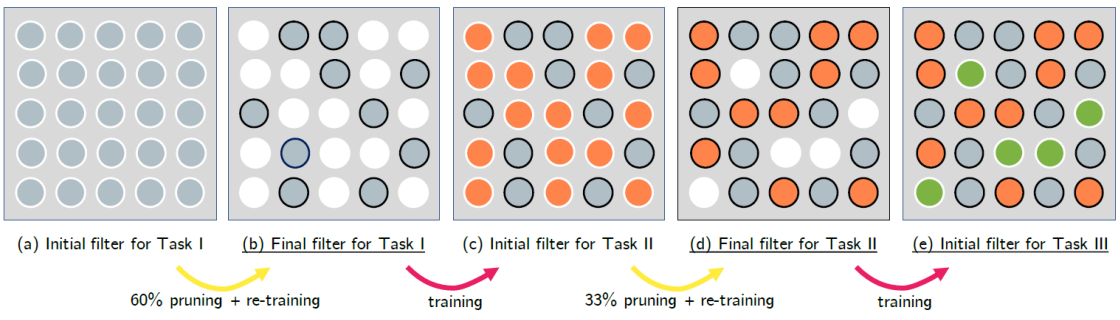

Figure 2.3. Graphical representation of packnet mechanism.

2.2.2 Piggyback

In this method [5], the main goal of the training procedure is to define a mask, with as many elements as the number of weights in the starting network, in order to select only the weights relevant for a given classification task at inference time. More specifically, in order to obtain these masks, the training procedure starts with a pretrained network and some real-valued numbers as elements of the matrix. These values are then trained through backpropagation by combining network binarization [6] and pruning [7]. In the end they are passed through a threshold function which will return only zeros and ones.

At inference time, before forwarding samples through the network, each weight is combined with the corresponding element of the matrix of the task the sample belongs to.

Adding these matrices introduces a very little overhead: just one bit for each weight of the network and the operation to combine each weight with one element of a matrix, which is a very fast operation.

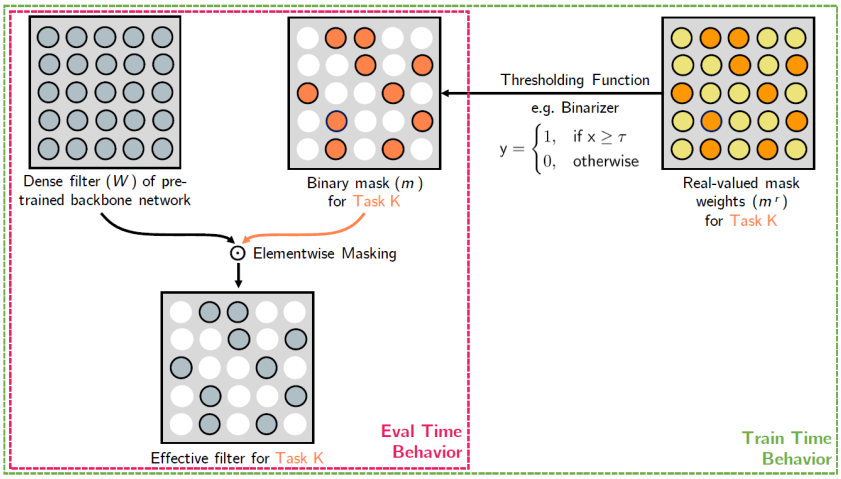

Figure 2.4. Graphical representation of piggyback mechanism.

2.3 Artificially generated samples

Approaches based on artificially generating samples of previous tasks to overcome the *catastrophic forgetting* phenomenon make use of deep generative models that go by the name of *Generative Adversarial Networks* (GANs) [8]. These models are able to maximize the likelihood of generated samples being in a real distribution and, in our cases, distributions of images. Within the GANs framework a zero-sum game between a generator G and a discriminator D is defined. The first is in charge to create artificial samples learning how to mimic the real data distribution as closely as possible, while the latter learns how to distinguish them from the real ones. In the equation 2.1 the objective function of this zero-sum game is illustrated:

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))] \quad (2.4)$$

Using such a technique, storing samples from previous tasks can be avoided or, at least, greatly reduced in order to reduce the memory footprint required by a given solution, as well.

On the other hand, the training time for a given task could still be a problem. In fact, we have to consider the time to generate artificial sample for all the so far seen classes and that, typically, the training set would be quite large since it could include a lot of samples for each class, regardless they are real or artificially generated.

2.3.1 Exemplar-Supported Generative Reproduction

For this method [9] the incremental learning meaning is slightly different from the others since it violates the first principle of the three ones defined

at the very beginning of this work: the model learns about a single class at a time, instead of a single task (that is a group of classes).

As a new class comes, the followed steps are represented in figure 2.6 and work as follows:

1. train a generator of artificial samples of that class that is nothing more than a GAN;
2. build a dataset by including both real samples of the new class and real and artificial ones (created through the past generators) of old classes;
3. train basically from scratch the network with the dataset of the previous point (similarly to *Jointly Training* trivial approach presented at the beginning of this work);
4. for each class, select the most representative samples and store them.

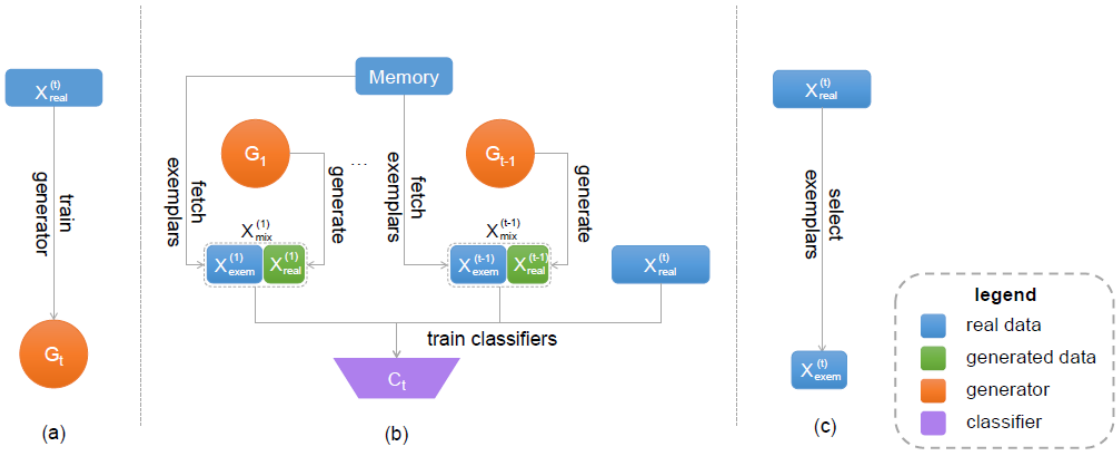

Figure 2.5. Training steps of *Exemplar-Supported Generative Reproduction for Class Incremental Learning.*

Similarly to *iCaRL*, the upper bound of the total number of samples to be stored is an input parameter, instead the metrics telling how much a sample

is representative is different: in this case the used metric is the prediction score of the right class made by the just trained network.
As anticipated before, this approach stresses the constraints about disk and, moreover, time resources since some samples of already seen classes are stored and for each new class the whole model is trained basically from scratch. The overhead introduced by this approach is much more relevant then others. First of all the whole procedure described above have to be executed for each new class instead of for a group of them, so much more frequently. Then, starting from the beginning of the procedure, GAN models requires time to be trained and disk space to be stored first and, in addition, a not negligible time to create the set of artificial samples. The worst aspect is, for sure, that the model should be trained from scratch for each new class and, of course, this training time increases with the number of different classes to learn. Least, but not last, there is the time to select, for each class, the most representative samples and the disk space to store them, that are relevant, as well.

2.3.2 Deep Generative Replay

This method [10] is based on the concept of *scholar* that is a tuple made of a *generator* and a *solver*. There is a scholar for each task, it is able to learn about the corresponding task and to acquire knowledge from the one of the previous task. This means that a scholar itself is able to transfer knowledge to another one, as well. This transfer of knowledge occurs from the $(i-1)^{th}$ task scholar to the i^{th} one and, in this way, the last scholar is automatically learning about all previous tasks and, in particular, about both the generation of artificial samples and the classification of all so far seen classes.

More formally, when a new task has to be learnt, the applied training procedure works as follows:

1. the last generator is used to create artificial samples about all previous tasks and these are joined together with the real samples of the new task into a single training set;

2. the training set output of the previous point is used to make the new generator to learn the distribution of both all old classes and the new ones;

3. the last scholar is used to create both artificial samples of old classes and their label by using its generator and its solver respectively;

4. data produced at the previous point together with new training data are used to train the new solver.

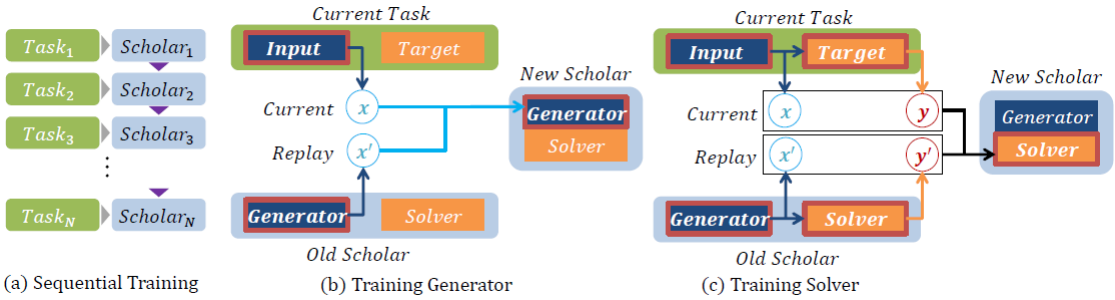

Figure 2.6. Training steps of *Continual Learning with Deep Generative Replay*.

This method is characterized by a non negligible overhead. This is especially true for the step of the generation of very many artificial samples together with the corresponding labels and, moreover, because, for each new task, the training of the solver is executed considering a lot of samples of all seen classes.

On the other hand the required amount amount of disk space is relatively little: no previous samples and, at most, two generators (the new one and the one just before of that) are stored.

Chapter 3

The chosen architecture

After a study of many different possible solutions to deal with the *incremental learning* problem, we decided to study in deep and implement the *Expert Gate: Lifelong Learning with a Network of Experts* [11] one. This work tries to avoid many little defects that most of the so far presented solutions unavoidably have due the way they are thought to work.

For example, for approaches that simply add output neurons for new classes and that make use of the distillation term into the training loss, there are, at least, three drawbacks. The first is risk of negative inductive bias when tasks are not related. The second is that the shared part of the network could fail in trying to learn information that are good for all involved tasks. The last is that for each new task the whole network is retrained.

Then, as widely explained in the previous chapter, the solutions based on masking some weights can learn a limited number of incremental tasks and, in order not to saturate the capacity of the network, they start from a very huge network.

At last, for all approaches based on the generation of artificial samples, the

main problem is the training time, since at each new task the network is trained from scratch. In addition it may not be able to capture knowledge that fits well all different tasks.

In addition to all just presented defects, all of the illustrated methods have a common problem: they could need a lot of space in memory to load all the needed models. According to the specific technique, it is due either to the dimension of the single used network or because additional modules, like generators, are needed together with the main network. Said that, we have also to consider that modern GPUs, used to speed up the training and testing of neural networks, has a limited amount of memory with respects to CPUs. Now, taking into account what we have just illustrated about the common problem of all previous method, in some cases, even the GPU memory could became a real constraint.

Let us think for example to an embedded system like a drone. It could not be made with too large hardware components and, as a direct consequence, the available memory could inevitably be a limitation. Now, let us thinking to a possible real application of this drone: it could be use to recognize objects in different rooms, objects that could be either outside or inside and even with different lighting conditions. It should be immediately clear how this is a problem that fits quite well the *incremental learning* approach. In fact we can think to each of the combinations of the different presented factors as a task that the algorithm inside the drone should learn.

Concluding this introduction, due to the limited memory of the drone, one could think to a way to train a different classifier for each of the tasks and, at inference time, to load only the needed one in memory in order not to saturate it. The core idea behind the chosen solution is something just like

this and it will be deeply examined in the next sections.

3.1 Expert Gate: Lifelong Learning with a Network of Experts

As widely explained before, the main idea consists in separating the tasks both for all just explained reasons and in order to deal with each task easily. More concretely the used architecture is the one depicted in figure 3.1 and

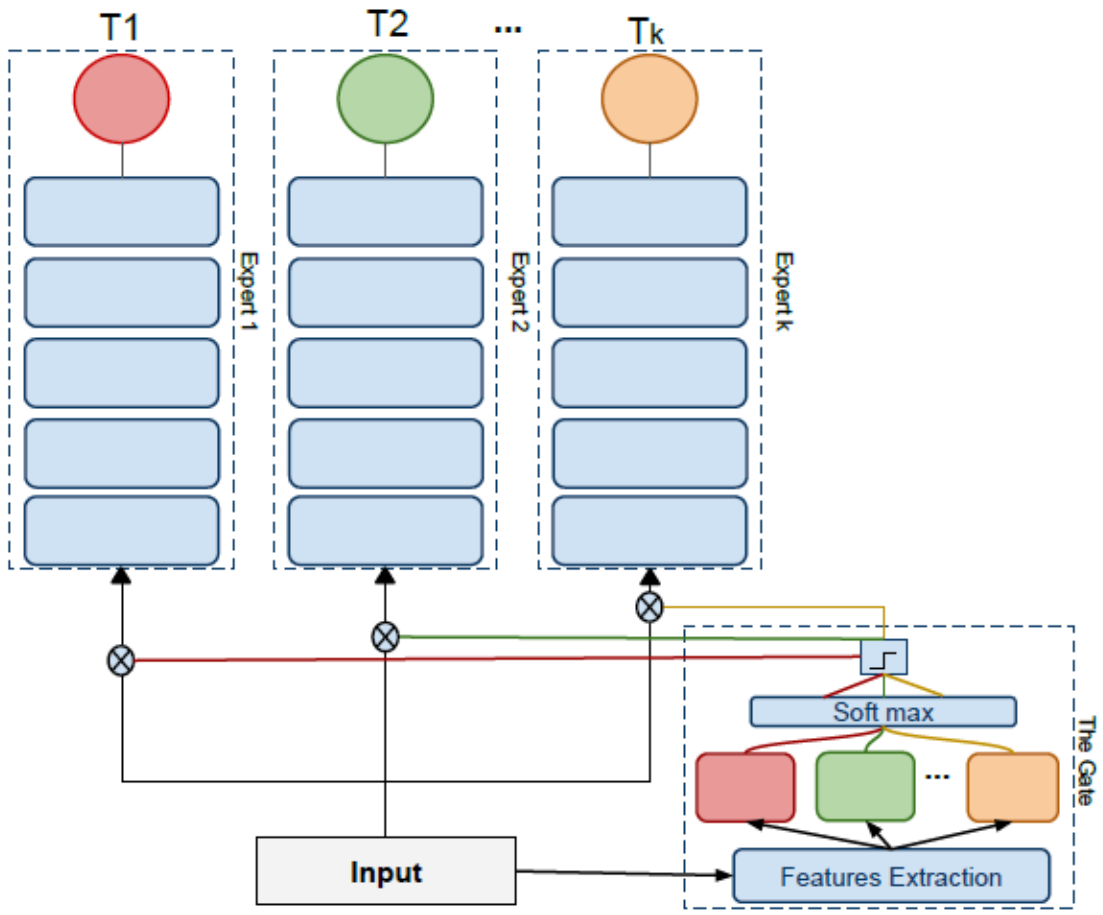

Figure 3.1. The whole architecture of *Expert Gate: Lifelong Learning with a Network of Experts.*

now we are going to present it. For each task two models are trained: a little *autoencoder* and an *expert*, that is a traditional classifier.
The first part of the architecture is made, in the following order, of a features extractor, as many autoencoders as the number of the seen tasks and a simple softmax function. These elements constitute a sort of gate: it receives a sample in input, predicts the task the sample belongs to and forwards it to the corresponding expert. The architecture continues with as many experts as the number of seen tasks and each of them works as traditional classifier for the classes of the corresponding task. Just for sake of simplicity, from now on, we will refer to this architecture with *AutExp*.

3.2 The Autoencoder Gate

An *autoencoder* [12] is a neural network that learns to produce an output similar to its input [13]. The network is composed of two parts: an *encoder* $f = h(x)$, which maps the input x to a code $h(x)$ and a *decoder* $r = g(h(x))$, that maps the code to a reconstruction of the input. The loss function $L(x, g(h(x)))$ is simply the reconstruction error.
In general two types of autoencoders could be used. When the encoder learns, through a hidden layer, a lower dimensional representation of the input, then the autoencoder is defined *undercomplete*. When instead, the encoder learns, through a hidden layer, a higher dimensional representation, then the autoencoder is defined *overcomplete*.
Applying and autoencoder can be thought very similarly to applying the PCA method for data dimensionality reduction [14]. In fact, a linear autoencoder, that is one with the *Euclidean loss* as objective function, subspace of the

PCA approach. However, non-linear autoencoders yield better dimensionality reduction compared to PCA. For this reason autoencoders have been preferred for the data dimensionality reduction purpose.

Autoencoders are usually used for data dimensionality reduction or even to learn feature representations in an unsupervised manner. Here, they are used for a different goal. The lower dimensional subspace learned by one of our undercomplete autoencoders will be maximally sensitive to variations observed in the task data but insensitive to changes orthogonal to the manifold. In other words, it represents only the variations that are needed to reconstruct relevant samples. Given all these reasons, we can formulate the main hypothesis this work is based on: the autoencoder of one domain/-task should be better at reconstructing the data of that task than the other autoencoders. At this point, comparing the reconstruction errors of the different tasks' autoencoders then allows to successfully forward a test sample to the most relevant expert network.

In figure 3.2 we have illustrated a part of the whole just depicted autoencoder gate and it is better to remember that the input of this architecture is not made of images but of a set of their features extracted using a certain model. In the original paper an *AlexNet* pretrained on *ImageNet* has been used as features extractor. In the current work we used a *VGG16* pretrained on *ImageNet* instead, since it has a very simple architecture and we thought that, for these reason, it would not have distorted too much the differences between images of different tasks which, then, would have allowed to well discriminate among the reconstruction of different autoencoders of each input samples.

Before actually passing the extracted features through the autoencoder, a

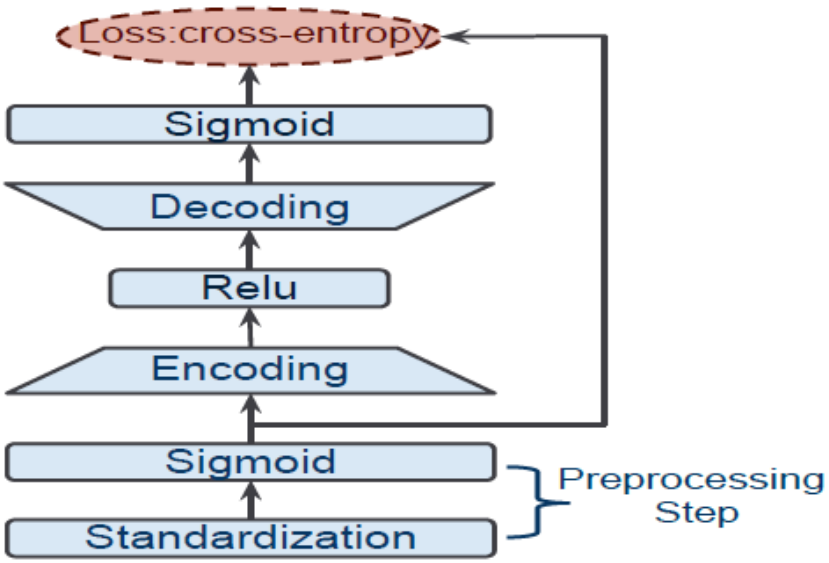

Figure 3.2. Part of the whole autoencoder gate structure.

preprocessing step is applied on them. The first operation is standardization which has the goal to increase the robustness of the hidden representation to input variations. Typically this operation is done using the statistics of data the network is trained on. Here the problem is that, since we have trained each autoencoder on a different training set, when, at inference time, we would compare the reconstruction errors of all autoencoders, they would have not been comparable. For this reason, the used statistics have been the same for all autoencoders and simply they were the mean and standard deviation of features extracted by our *VGG16* model given the *ImageNet* dataset in input. Just after the standardization, a simple *sigmoid* activation function is applied in order to map each input value into the [0, 1] interval. The actual autoencoder starts with the *encoding* layer, that is nothing more than a fully connected layer and that is followed by a *ReLU* [15] activation function which introduces sparsity in the hidden units and leads to a better

generalization. The *decoding* layer is, again, a fully connected layer with as many output units as the number of the input ones and that is followed by a *sigmoid* activation function. In this case, this last activation is needed because of the used loss to train the autoencoder that is the *cross entropy* one which requires input values to be into the [0, 1] interval.

3.3 Predicting the task's expert

At test time, in order to select the task the input sample belongs to, and so, the corresponding expert that will be used as classifier, that sample is reduced in dimensionality, preprocessed and, then, forwarded to all available autoencoders. Then, combining the input and the corresponding output array, we can compute the reconstruction error of each autoencoder. In particular, this error is computed simply as *Euclidean distance*. At his point, a *softmax* layer is applied with all reconstruction error values in order to retrieve, for each task, a value representing a sort of probability that the input sample belongs to that task. In particular, for the i^{th} reconstruction error value, the corresponding probability value is computed as in the equation 3.1:

$$p_i = \frac{e^{-err_i/t}}{\sum_j e^{-err_j/t}} \tag{3.1}$$

In the just illustrated equation the t is the *temperature* values and it has been set to 2 as in [16] and [17] works.

Once that the probabilities have been computed, only the expert that corresponds to the maximum values among them is loaded in memory to work as a traditional classifier. In some cases one could even consider more than

one expert if the highest probability values are very close and then analyze the different predictions.

3.4 The experiment

The just described architecture has been tested on the same datasets explored in the reference paper but one and they are: MIT Scenes [35], Caltech-UCSD Birds [22], Oxford Flowers [24], Stanford Cars [23], and FGVC Aircrafts [36]. These datasets involve classification problems of 67, 200, 102, 196, 90 classes respectively. We have considered a common data size of 256x256 pixels for all images (3 channels), the 25% of each dataset as test set and the 25% of the remaining samples as validation set. As already said we used a *VGG16* as features extractor, while we used a *InceptionResnetV2* [46], pretrained on ImageNet, as base model for the experts. The choice of that base model for the experts has been characterized by the fact that, for each dataset, there were quite many different classes to learn, so we needed a network architecture comlplex and deep enough. Here we have the results of the experiment and both the accuracy of the autoencoders gate at predicting the correct task ans the final classification accuracy are shown in figures 3.3 and 3.4 respectively.

Looking at the charts, it is extremely evident how well the autoencoders gate performs. In fact most of the misclassifications were made by the experts. Anyway, this experiment well demonstrates a solution for the incremental learning since the architecture was very good at discriminating classes tasks. Of course these results could be improved just working on the experts (i.e. changing the base model, exploring a wider space of parameters for the cross

validation of them, etc.).

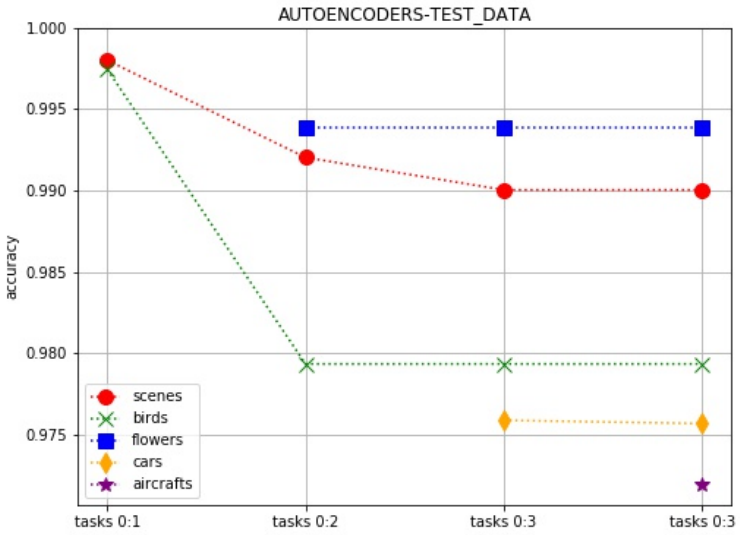

Figure 3.3. Autoencoders tasks accuracy increasing the number of tasks.

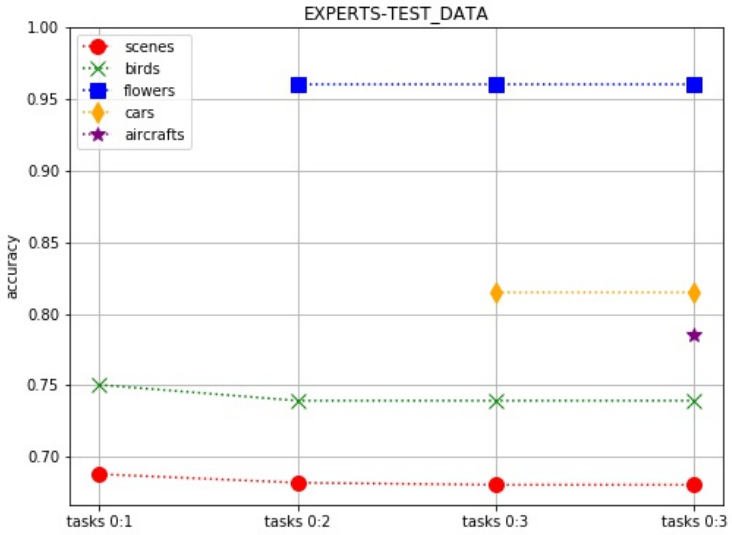

Figure 3.4. Final classification accuracy increasing the number of tasks.

3.5 A new application: CIFAR-100

As one could see from the results of the experiment in the previous section, the autoencoders gate is extremely good at predicting the belonging task of an input sample at inference time. Despite this, we have to admit that we have considered classes of the same domain for each task and the real reason behind the more than satisfying performance of the autoencoders could be just this. In fact, since images of the same task are in some way similar, acquiring the knowledge to recognize samples of a single task could be quite easy for an autoencoder.

In this new section we are going to present a new experiment in which the autoencoders gate has been stressed much more than before in order to see if it is actually able to learn how to recognize different taks. Here we have considered each task made of some randomly chosen classes of the CIFAR-100 dataset. Of course, randomly choosing the classes of each task is important in order to have classes of different domains.

The base model has been the *VGG16* pretrained on *ImageNet* for both the features extractor and the experts. By the way the features extractor network was not used entirely but we cut it 3 layers before the end. The reason of that was that, considering the whole architecture we ended up in too few features to well capture the differences among samples of different tasks. Furthermore the expert base model was constituted by a much simpler network than before just because the classification problems were easier and involved less different classes. We have conducted two different experiments varying the number of tasks (5 and 10) and, as a consequence, the number of different classes per tasks (20 and 10). In figures 3.5 and 3.6 the mean accuracy

and its standard deviation as the number of tasks increases are represented. Examining these results it is rather palpable how the architecture is stressed

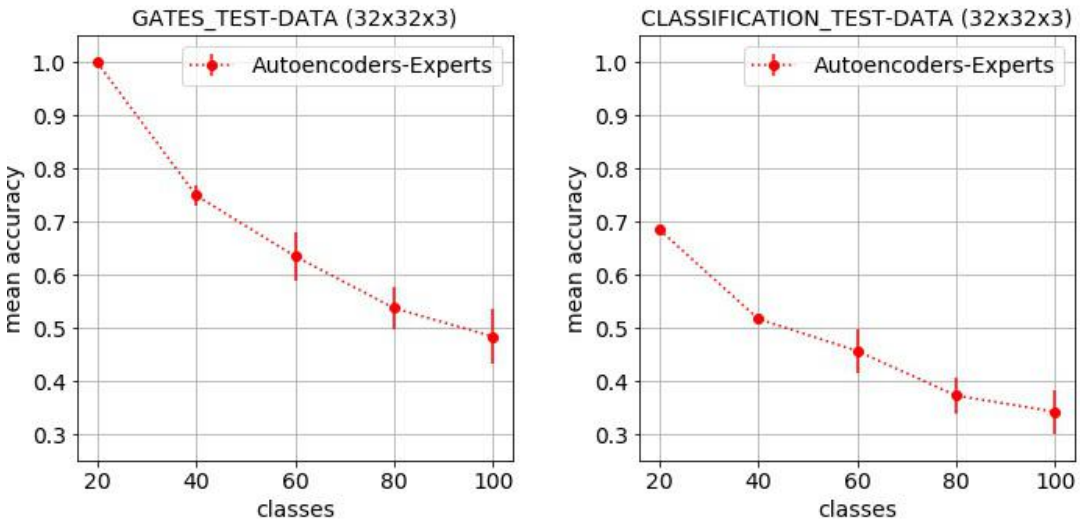

Figure 3.5. Gate and Classification accuracy for 5 tasks with data size 32x32x3.

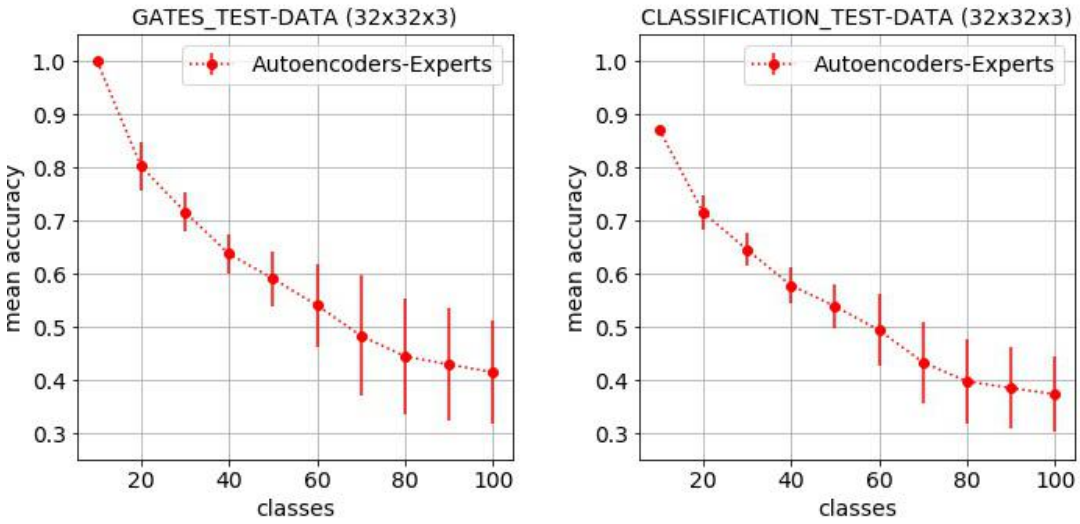

Figure 3.6. Gate and Classification accuracy for 10 tasks with data size 32x32x3.

by this new dataset. In fact, apart from the misclassifications of the experts, many mistakes are made by the autoencoders gate. In other words, our

supposition about the fact that considering images of the same domain for each task makes easier the work of the autoencoders gate has been confirmed. A first idea to improve these results was to consider the same data but with a higher data size. This idea came from the willing to try the behaviour of the autoencoders giving them more features to reconstruct. So, starting from the original data size of the CIFAR-100 samples of 32x32x3, we got samples with size of 128x128x3 by interpolating pixels. In figures 3.7 and 3.8 the results of this new experiment are shown.

We can see how the performance improved with the higher data size even if they are not comparable with the ones of the experiment with each task containing images of the same domain.

In the next chapter two new approaches based on *Bayesian* models will be proposed and we will compare all architectures even increasing the number of tasks the whole dataset is splitted into.

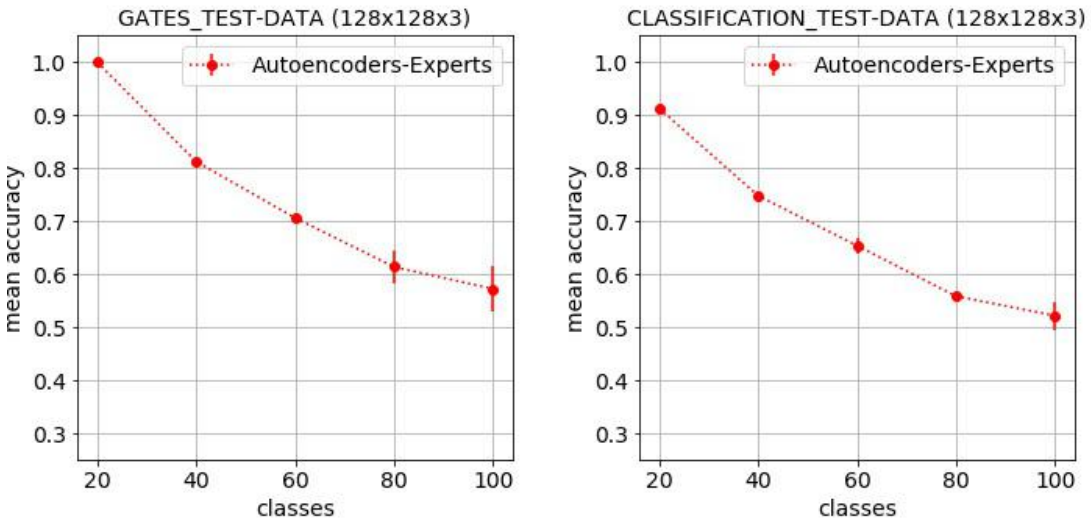

Figure 3.7. Gate and Classification accuracy for 5 tasks with data size 128x128x3.

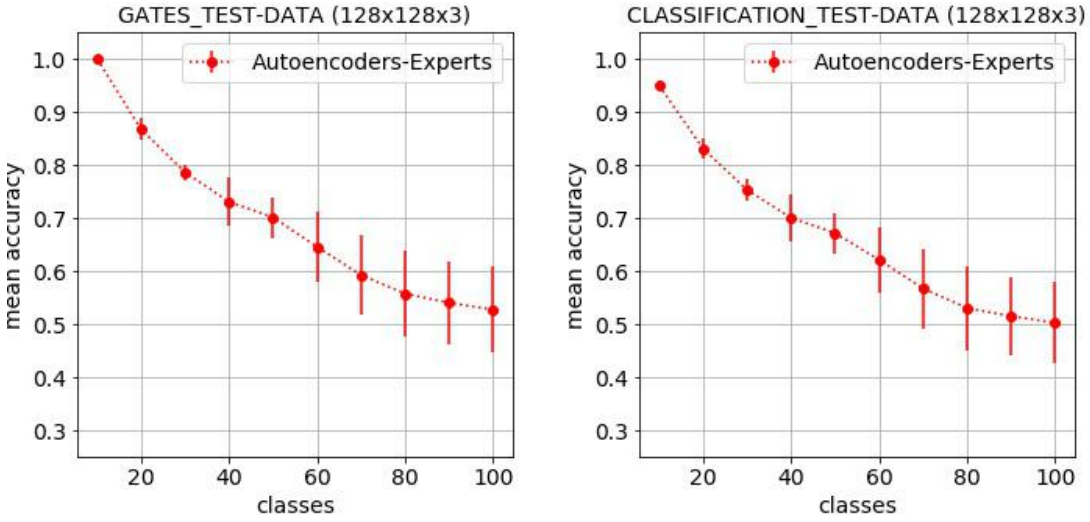

Figure 3.8. Gate and Classification accuracy for 10 tasks with data size 128x128x3.

Chapter 4

The Bayesian approach

In the previous chapter we have demonstrated that the autoencoders gate performs extremely well in case of images of the same domain for each task, but this is not the case in the opposite case. Here we want to introduce bayesian models in order to improve the task prediction performance just in the second case.
Very briefly, a bayesian model is a specific neural network that, given an input sample, is able to compute the uncertainty of its class prediction on that sample. Said that, we used the same previous architecture but in two new variants which will be referred with *BayExp* and *AutBay* respectively:

- A gate of *Bayesian* models followed by the same number *Experts*;
- A gate of *Autoencoders* followed by the same number of *Bayesian* models (which are both in charge of correcting the gate tasks predictions and of running the final classification).

Just like for autoencoders, each bayesian model is trained using only data of one task and, at inference time, we rely on this in order to well predict the

belonging task of each sample. In fact, a bayesian which has never seen a certain class should produce a higher value of uncertainty. Exploiting this typical characteristic of bayesian neural networks, better predicting the task of each input sample or correcting the autoencoders predictions have been possible in some cases.

4.1 Introduction to Bayesian neural networks

Tools such as neural networks and convolutional neural networks are very well known and extensively used in a deterministic form: forwarding as many times as we want the same sample through the network, it will produce always the same prediction. Typically the last layer of a network is a *softmax* function which gives, for each class, a value representing how much confident the network is about the fact that the sample belongs to that class. Unfortunately, this way of interpreting the output of the network is not totally correct. In fact, it could happen that, even if the highest *softmax* output is relatively much higher than others, the model could still be uncertain about that prediction. Another important defect of the *softmax* function is that it always returns predictions, even if we present to the model a sample of a class it has never seen before.

Now, looking at real-world applications, handling the uncertainty of predictions could be very useful: let us think to a case in which the network says to be uncertain enough of a prediction, then an human being could intervene. Situations like this one can happen, for example, in a post office, sorting letters according to their zip code, or in a nuclear power plant with a system responsible for critical infrastructure [38].

4.2 Dropout approximation

Bayesian probability theory offers us mathematically grounded tools to build bayesian network and its uncertainty, but these usually come with a prohibitive computational cost. The main idea of these tools is to associate a *Gaussian* curve for each weight of the network and then, during the forwarding of the input through the network, the actual value of each weight is sampled from the corresponding *Gaussian*. This behaviour has two main consequences: forwarding many times the same samples through the network, it may produce different outputs and, instead of optimizing weights during training, mean and variance of each of those *Gaussians* are optimized.
In [39] has been recently showed that a typical optimization of dropout neural networks is equivalent to Bayesian learning via variational inference with a specific variational distribution. This means that a neural network with arbitrary depth and non-linearities, with dropout applied before every weight layer, is mathematically equivalent to an approximation to the probabilistic deep *Gaussian* network. The fundamental advantage of this dropout approximation, with respect the traditional Bayesian model, is that the computational cost is not prohibitive anymore.
A dropout layer works randomly cutting some connections (the corresponding weight is set to 0) between nodes of two consecutive layers. The percentage of cut connections depends on the chosen dropout rate which is an input parameter and represents, for each node, the probability to be cut. Traditionally dropout layers have been used in order to avoid overfitting and to make the network to generalize at inference time, with the final goal of increasing the its accuracy on test data. More specifically, these layers are activated only

during the training phase in order to make the learning process more complex for the network, while they do not cut any weight when testing. In this way, if the network was able to accomplish a good performance without the cut weights, then it will very likely perform even better exploiting all its weights. In the approximation of a Bayesian network, dropout layers are always ac-

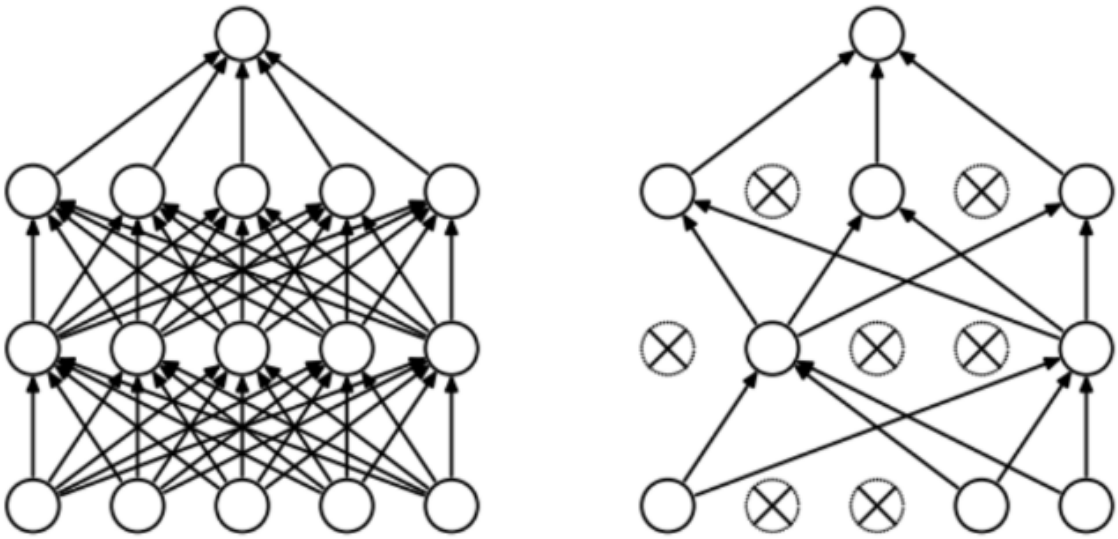

Figure 4.1. Traditional application of dropout: inference on the left and training on the right.

tivated and, remembering that cutting nodes is a random operation, then every time the same sample is forwarded through the network, the output could change depending on which nodes has been cut.

4.3 Uncertainty analysis

The variability of the output of a Bayesian network just presented in the previous section is main element exploited to compute the uncertainty of the network itself about its predictions. In this work we have used the method proposed by [40] for the computation of the model uncertainty. According

to this approach the whole uncertainty is made of two terms: *epistemic* and *aleatoric* ones.

The *aleatoric* component captures noise inherent in the observations. It is useful in big data scenarios where the epistemic component of uncertainty disappears, potentially leading the model to overconfident predictions. Moreover, it provides a measure of the diversity of the input data, it can help to differentiate between inputs that are easier to classify or not, based on their ambiguity.

The *epistemic* component captures the actual uncertainty of the model. It is useful when big data sets are not available because these are the scenarios that require to express a high uncertainty for the model parameters. Furthermore, it is key to understand out-of-distribution samples, observations that are different from the training data. This can be particularly useful for critical applications where it is not possible to train the model on the full distribution of input data. For instance, considering an autonomous driving system, it is not possible to train the neural network on all possible scenarios, therefore the autopilot should notify the human driver to take control in the presence of uncertain images from the street.

4.3.1 Uncertainty estimation

Let $D = \{(x_i, y_i)\}_{i=1}^{N}$ be a realization of independently and identically distributed variables with $x_i \in R^d$ and $y_i = (y_i^{(1)}, ..., y_i^{(K)}) \in \{0,1\}$ are the i^{th} input and the corresponding one-hot encoded output respectively. In addition N is sample size, d is the dimension of the input variables and K is the number of different classes. Let us consider now a Bayesian neural network represented by $\omega \in \Omega$, that is the vector of network parameters (weights

and biases). Given the prior distribution of ω we can compute the posterior distribution and the predictive one for a new input x^* and a new output $y*$ (4.1 and 4.2 respectively).

$$p(\omega|D) = \frac{p(D|\omega)p(\omega)}{p(D)} = \frac{\prod_{i=1}^{N} p(y_i|x_i,\omega)}{p(D)} \tag{4.1}$$

$$p(y^*|x^*, D) = \int_{\Omega} p(y^*|x^*,\omega)p(\omega|D)d\omega \tag{4.2}$$

Now denoting the last K-dimensional pre-activated linear output of the previous neural network by $f^{\omega}(x) = (f_1^{\omega}(x), ..., f_K^{\omega}(x))$, then the predictive probability is the one at 4.3.

$$p\{y^{(k)} = 1|x,\omega\} = p\{y^{(k)} = 1|f^{\omega}(x)\} = \frac{e^{f_k^{\omega}(x)}}{\sum_{j=1}^{K} e^{f_j^{\omega}(x)}} \tag{4.3}$$

Despite the simplicity of this formalization, the learning is still a complex issue due to the computation of the posterior $p(\omega|D)$ which requires an integration with respect to the whole parameters space Ω and this does not have a closed form solution typically.

Different methods to overcome this problem have been proposed over time with more or less success. [41] proposed a *Laplace* approximation of the posterior but it was a too poor one. [42] proposed a *Markov Chain Monte Carlo* sampling approach using *Hamiltonian* dynamics which led to a set of posterior samples without directly computing it actually, but it was still a computationally prohibitive technique.

A more recent alternative was proposed by [43] and [44] approximating the posterior distribution with a tractable variational one $q_\theta(\omega)$ indexed by a variational parameter θ. The optimal variational distribution is the closest distribution to the posterior among the pre-determined family $Q = \{q_\theta(\omega)\}$. The closeness is often measured by the *Kullback-Leibler* divergence between

$q_\theta(\omega)$ and $p(\omega|D)$. In this way they moved from an integration problem to an optimization one, which could be tackled with online learning, but whose quality depended on the family of distributions Q.

This work is totally based on the paper [40] which follows the idea to split the uncertainty into the *aleatoric* and *epistemic* components. Now let us see how they derived the definition of both of them.

Let $\hat{\theta}$ be the optimized variational parameter iteratively minimizing Monte Carlo approximated version of the equation 4.1 so, at inference time, the variational predictive distribution, which approximates the predictive distribution, can be written as

$$p_{\hat{\theta}}(y^*|x^*) = \int_\Omega p(y^*|x^*, \omega) q_{\hat{\theta}}(\omega) d\omega \tag{4.4}$$

which is estimated by

$$\hat{q}_{\hat{\theta}}(y^*|x^*) = \frac{1}{T} \sum_{t=1}^{T} p(y^*|x^*, \hat{\omega}_t) \tag{4.5}$$

This estimator converges in probability considering a set of realized vectors $\{\hat{\omega}_t\}_{t=1}^T$ randomly drawn from the variationl distribution $\hat{q}_{\hat{\theta}}(\omega)$ (T is the number of samples). The variance of the variational predictive distribution $\hat{q}_{\hat{\theta}}(y^*|x^*)$ is given by the following equation (for further explanations about the single operations of this equation, please take a look at the reference paper previously indicated).

$$\begin{aligned} Var_{\hat{q}_{\hat{\theta}}(y^*|x^*)}(y^*) &= E_{\hat{q}_{\hat{\theta}}(y^*|x^*)}\{y^{*\otimes 2}\} - E_{\hat{q}_{\hat{\theta}}(y^*|x^*)}(y^*)^{\otimes 2}\} \\ &= \underbrace{\int_\Omega [diag\{E_{p(y^*|x^*,\omega)}(y^*)\} - E_{p(y^*|x^*,\omega)}(y^*)^{\otimes 2}] q_{\hat{\theta}(\omega)d\omega}}_{aleatoric} \\ &+ \underbrace{\int_\Omega \{E_{p(y^*|x^*,\omega)}(y^*) - E_{\hat{q}_{\hat{\theta}}(y^*|x^*)}(y^*)\}^{\otimes 2} q_{\hat{\theta}(\omega)d\omega}}_{epistemic} \end{aligned} \tag{4.6}$$

In [45], starting from the previous definitions of the two types of uncertainty, a concrete method to estimate those quantity has been developed. The used model has been a Bayesian neural network with the last layer before activation that computes the mean and variance of logits. Now, let $f^{\omega}_{kendall}(x^*) = (\mu, \sigma^2)$ be the $2K$-dimensional pre-activated linear output of the neural network with μ and σ^2 the mean and variance of the K nodes, for the realized vectors $\{\hat{\omega}_t\}_{t=1}^{T}$, they proposed the following estimator for the uncertainty

$$\underbrace{\frac{1}{T}\sum_{t=1}^{T} diag(\hat{\sigma}_t^2)}_{aleatoric} + \underbrace{\frac{1}{T}\sum_{t=1}^{T}(\hat{\mu}_t - \overline{\mu})^{\otimes 2}}_{epistemic} \tag{4.7}$$

where $\overline{\mu} = \frac{1}{T}\sum_{t=1}^{T}\hat{\mu}_t$.

Estimators at 4.7 have some limitations when they are used for estimating the classification uncertainty. First, they do not consider the predictive probabilities, but they model the variability of linear predictors only, without taking into account that the covariance matrix of a multinomial random variable is a function of the mean vector. Second, the *aleatoric* uncertainty does not reflect correlations due to a diagonal matrix modeling.

Due to these limitations, some variations have been introduced so that the final definition of estimators for *aleatoric* and *epistemic* uncertainties became the following one

$$\underbrace{\frac{1}{T}\sum_{t=1}^{T} diag(\hat{p}_t) - \hat{p}_t^{\otimes 2}}_{aleatoric} + \underbrace{\frac{1}{T}\sum_{t=1}^{T}(\hat{p}_t - \overline{p})^{\otimes 2}}_{epistemic} \tag{4.8}$$

where $\overline{p} = \frac{1}{T}\sum_{t=1}^{T}\hat{p}_t$ and $\hat{p}_t = p(\hat{\omega}_t) = softmax\{f^{\hat{\omega}_t(x^*)}\}$. Now the computation of these estimators does not involve the σ^2 term and the number of parameters and operation has been reduced. Last, but not least, they converge in the probability at 4.6 as T increases.

Just to clarify, here we present a snippet of Python code representing the function that implement the estimator at 4.8. It takes as input a Bayesian model, the data and the sampling number and returns the two just described uncertainty components.

```python
import numpy as np

def compute_predictions_uncertainty(bayesian_model, X, T):
    p_hat = []
    for t in range(T):
        p_hat.append(bayesian_model.predict(X))
    p_hat = np.array(p_hat)

    aleatoric = np.mean(p_hat * (1 - p_hat), axis=0)
    epistemic = np.mean(p_hat ** 2, axis=0)
                - np.mean(p_hat, axis=0) ** 2
    predictions = np.mean(p_hat, axis=0)

    aleatoric_unc = np.zeros(len(X))
    epistemic_unc = np.zeros(len(X))
    for i, x in enumerate(X):
        predicted_class = np.argmax(predictions[i])
        aleatoric_unc[i] = aleatoric[i, predicted_class]
        epistemic_unc[i] = epistemic[i, predicted_class]

    return aleatoric_unc, epistemic_unc
```

Listing 4.1. Function to compute uncertainty of a model.

4.4 Two new proposed architectures

As anticipated in the introduction of this section, we developed two new approaches exploiting *bayesian* neural networks. In particular, we trained a bayesian model for each task and then, combining them alternatively with the autoencoders and the experts, we built two apparently similar architectures.

The first one is *BayExp* an it is analogue to the main architecture presented in this work. In fact the part of the experts is the same, but the autoencoders gate is replaced by a bayesians one. In particular, at inference time, the same sample is given to all bayesians (there is one for each tasks), each of them computes the uncertainty value for that sample and the predicted task will be the one corresponding to the least uncertain bayesian. Once predicted the task of the input sample, the classification part is exactly the same of before.

For the second architecture, that is *AutBay*, we have again an autoencoders gate, but they are followed by the same number of bayesian models this time. At inference time, the input sample is given to all autoencoders and, running the usual procedure, its belonging task is predicted. Now the sample is forwarded to the bayesian model which corresponds to the predicted task and it computes both the class prediction and the uncertainty value. If the uncertainty is below a given threshold, then we consider the prediction of the current bayesian as the good one, otherwise we send the sample to all existing bayesians. Each one computes both the predicted class and the uncertainty but only the prediction of least uncertain model will be actually considered. The uncertainty threshold of each bayesian is computed analyzing the uncertainty discrete distribution of the training samples. First, by exploiting

the *Otzu*'s algorithm [47] the value of uncertainty splitting the distribution in two modes is found. Then, starting from that value, we look for the one corresponding to the maximum delta in the distribution (for the mode with the highest values of uncertainty of course). In figure 4.2 we have an example of this procedure.

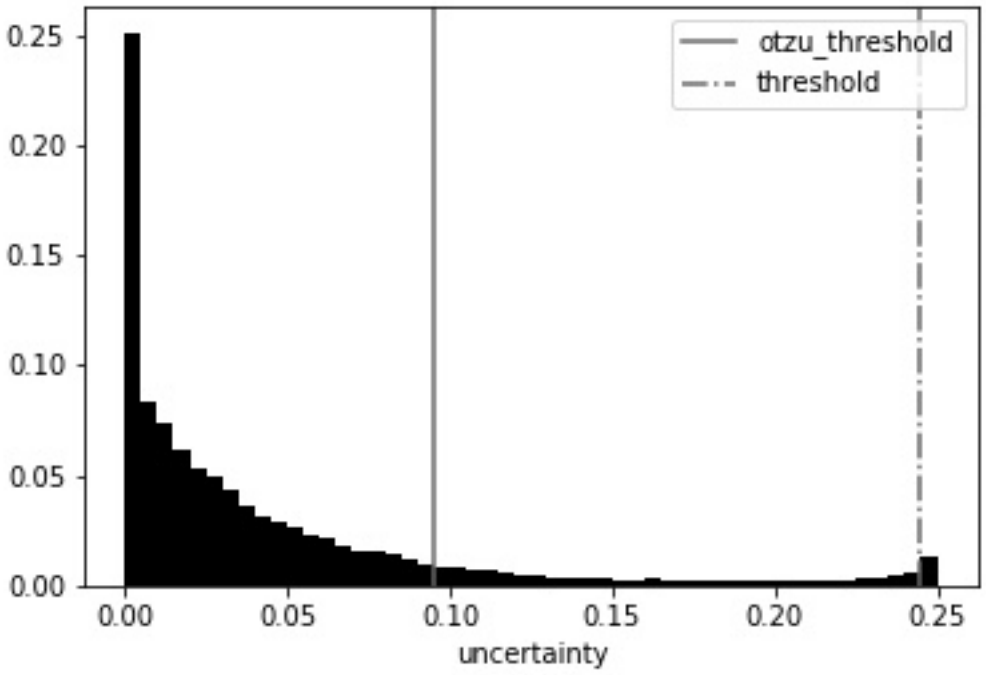

Figure 4.2. Example of how the threshold for bayesian models is found. The non-continual line represent the value we look for.

4.5 The experiment

In figures 4.3, 4.4, 4.5, 4.6 and 4.7 there are the results of these new architectures on CIFAR-100, including the traditional one, as well. We have considered 2, 5, 10, 20 and 50 tasks.

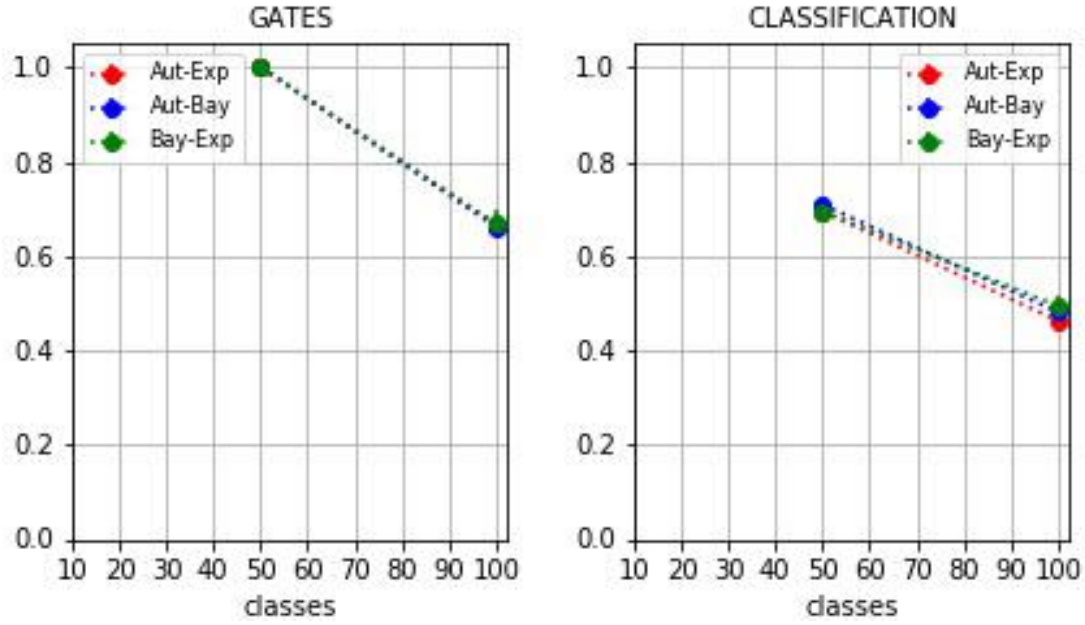

Figure 4.3. Gate and Classification accuracy for 2 tasks.

Those charts say that the two new approaches better perform than the traditional one as far as the number of tasks is relatively small. In fact when considering 20 and 50 tasks performances significantly degrade for the architecture with bayesians and Experts, while the architecture with autoencoders and bayesians performs quite similarly to the traditional one.

In parallel with respect to the previous analysis, now we look at the efficiency in terms of time for the training and, in particular, for the inference of the different architectures. For the training phase both the new architectures require more time than the first one simply because, for a new task, we have to train a bayesian CNN instead of an autoencoder or an expert and in both cases the training time is longer. For the inference phase we have the same situation because computing the uncertainties and the predictions of a bayesian model require passing the same data different times through it

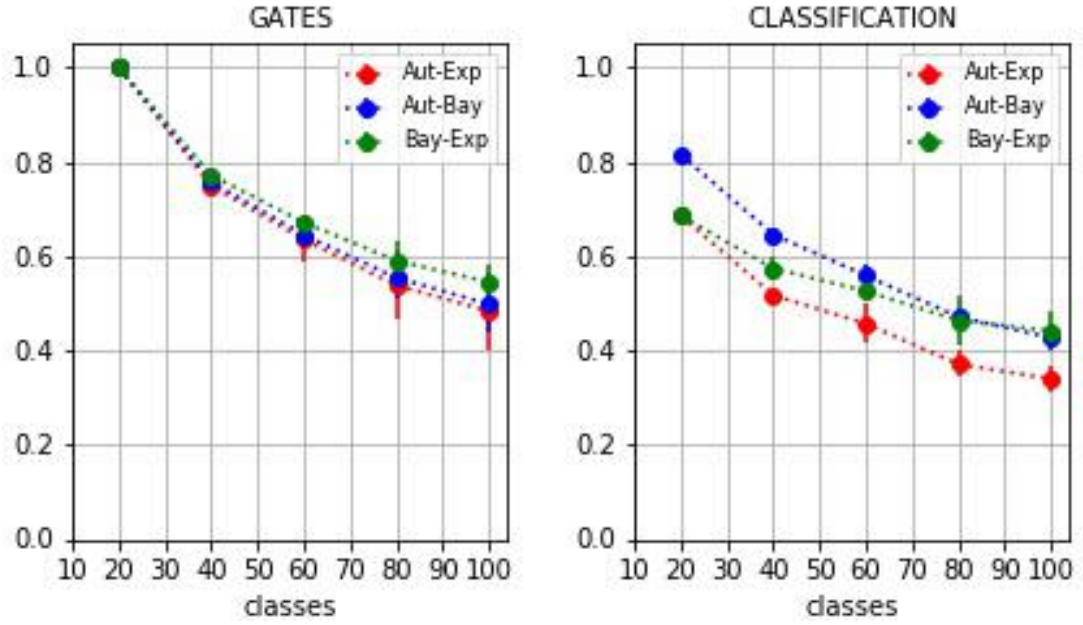

Figure 4.4. Gate and Classification accuracy for 5 tasks.

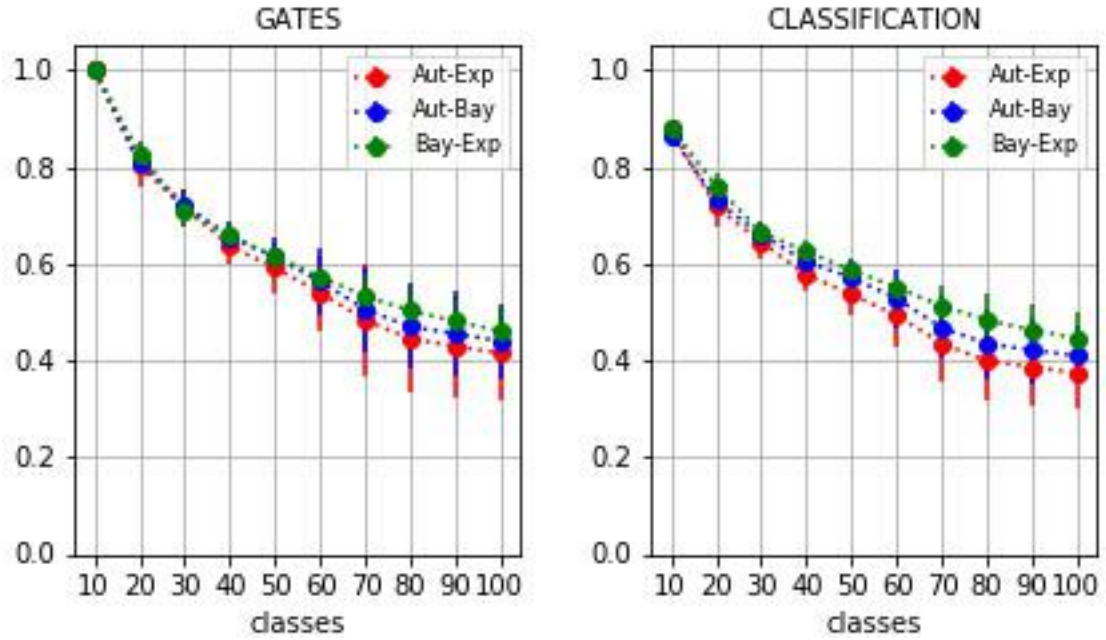

Figure 4.5. Gate and Classification accuracy for 10 tasks.

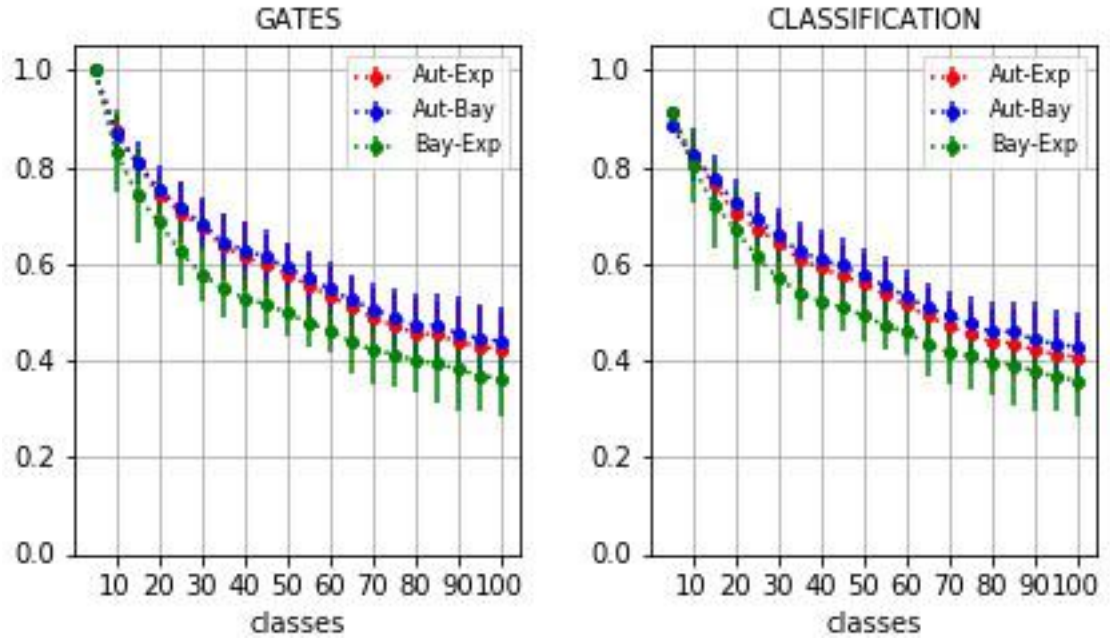

Figure 4.6. Gate and Classification accuracy for 20 tasks.

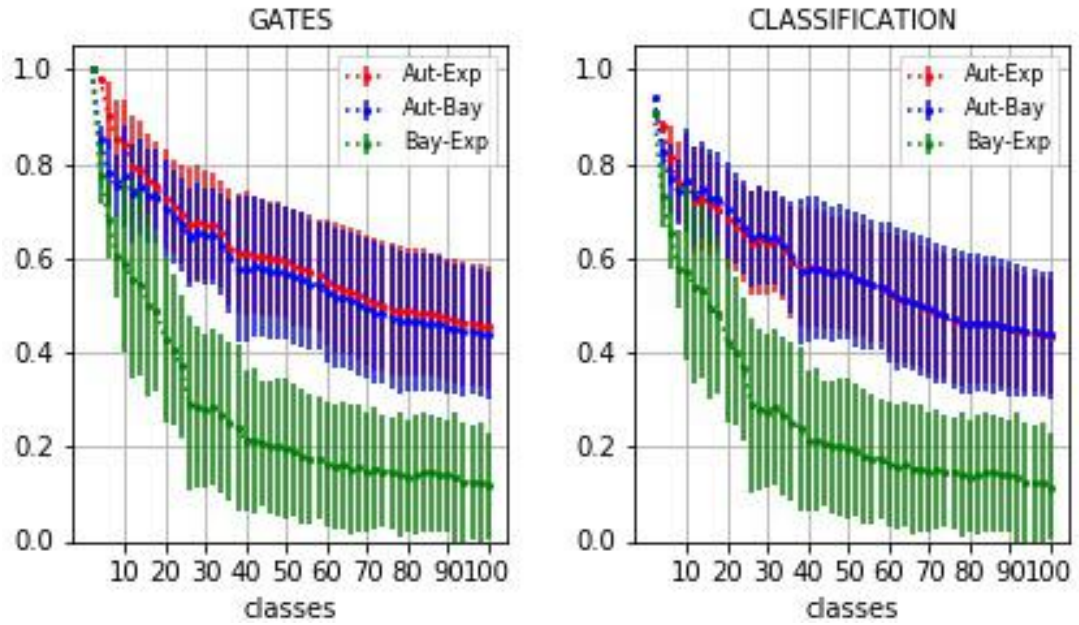

Figure 4.7. Gate and Classification accuracy for 50 tasks.

(100 times in our case). In figures 4.8, 4.9, 4.10, 4.11 and 4.12 there are some charts which compare the mean inference time per task of the three methods as the number of tasks increases.

It is evident that, as the number of tasks increases, the mean inference time of the architecture with autoencoders and experts is quite negligible with respect to the other two architectures. To be honest, having such a long inference time does not happen with many of the other state-of-the-art approaches and it is an aspect to be taken into account according to which are the actual needs of an application. By the way, at least the part of the inference procedure to predict the task could be made faster parallelizing the computation of actually used metrics to made this prediction.

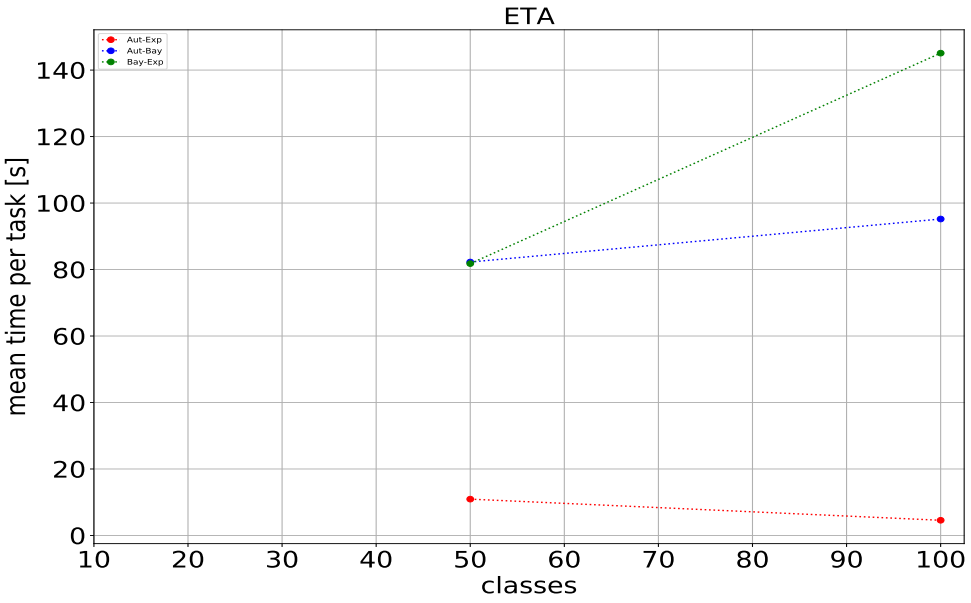

Figure 4.8. Mean inference time per task for 2 tasks.

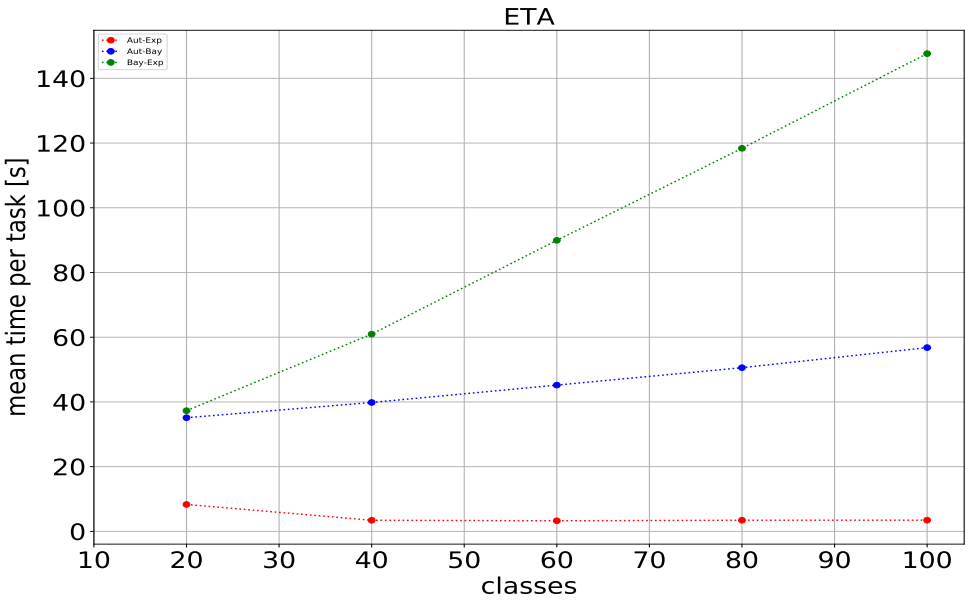

Figure 4.9. Mean inference time per task for 5 tasks.

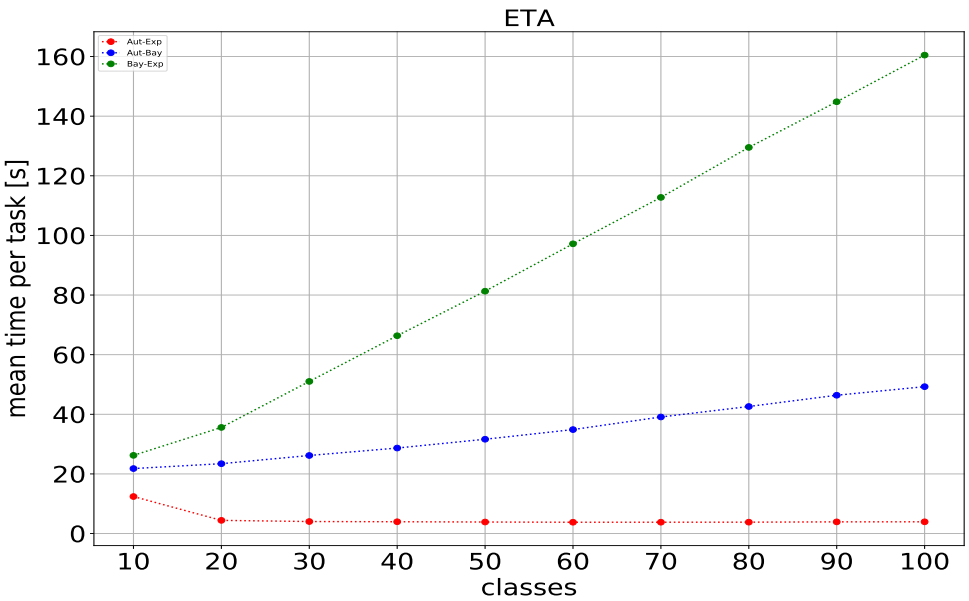

Figure 4.10. Mean inference time per task for 10 tasks.

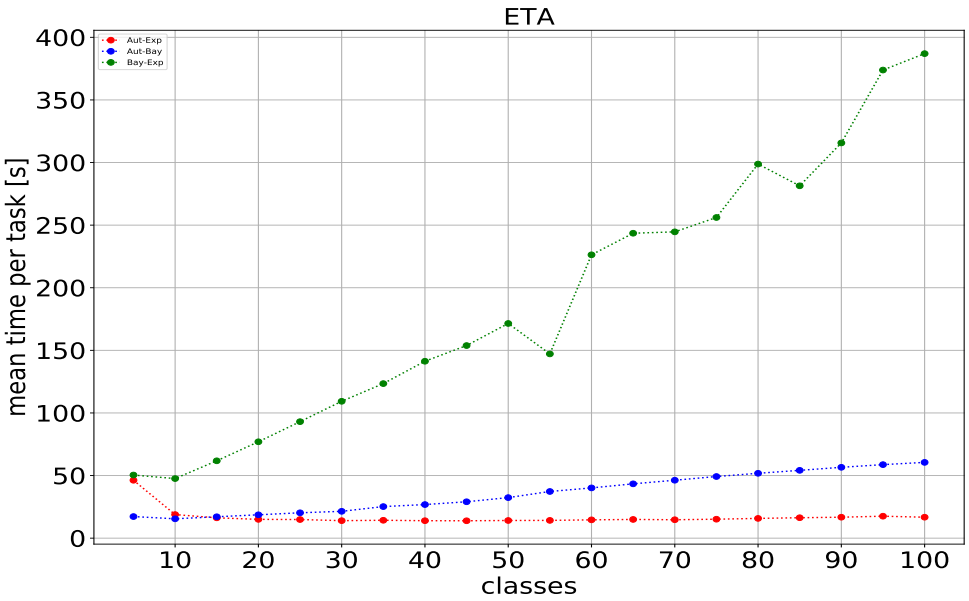

Figure 4.11. Mean inference time per task for 20 tasks.

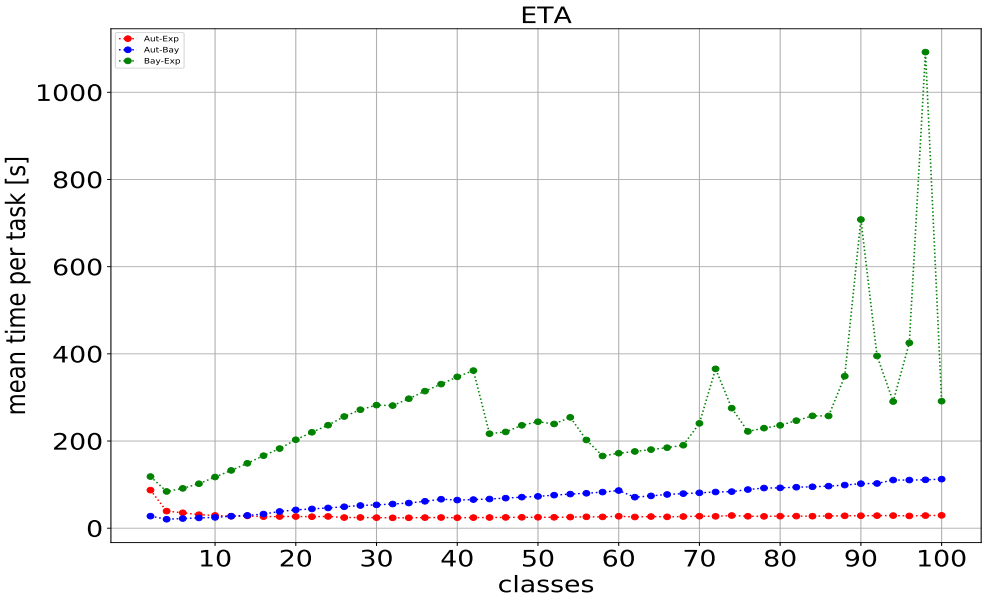

Figure 4.12. Mean inference time per task for 50 tasks.

Chapter 5

Conclusions

We have explored very many existing solutions for the incremental learning problem until we created two new solutions on our own. They both appeared very competitive and, in some cases, even better than some of the available state-of-the-art approaches. This is evident simply looking at our results, but it is better to highlight some apparently secondary aspects regarding the resources involved in our solutions. The most evident aspect is that they do not need to keep samples of the previous tasks which implies no specific requirements for the disk space (and probably less training time). Another interesting plus of our solutions is that even the actual memory could not represent a huge requirement since, at inference time, we can load in memory a model at a time (of course the time for the inference phase will be negatively affected by this). Last, but not least, the training time could be reduced training, for each task, the two needed models (i.e. autoencoder and expert or bayesian and expert) in parallel, since these steps are completely independent.

Despite these interesting aspects, we have also to mention that the inference

time could be a problem with our two new architectures.
In conclusion, let us see some of the possible future developments:

- increase the number of task;
- increase the hidden layers of the autoencoders;
- for the configuration with autoencoders and bayesians, add the final classification step run by experts;
- change the base model for the features extractor, bayesians and classifiers.

Appendices

Appendix A

iCaRL

A.1 Pseudo code

Here we have the pseudo code of the different parts of this method.

Algorithm 1 iCaRL - Classification

input x # image to be classified
require $P = (P_1, ..., P_t)$ # class exemplar sets
require $\phi : X \to R^d$ # feature map

for $y = 1, ..., t$ **do**
  $\mu_y \leftarrow \frac{1}{|P_y|} \sum_{p \in P} \phi(p)$ # mean-of-exemplars
end for
$y^* \leftarrow \underset{y=1,...,t}{\operatorname{argmin}} ||\phi(x) - \mu_y||$ # nearest prototype

output y^* # class label

Algorithm 2 iCaRL - Incremental train

input $X^s, ..., X^t$ # training examples in per-class sets
input K # memory size
require Θ # current model parameters
require $P = (P_1, ..., P_t)$ # current exemplar sets

$\Theta \leftarrow UpdateRepresentation(X^s, ..., X^t; P, \Theta)$
$m \leftarrow K/t$ # number of exemplars per class
for $y = 1, ..., s-1$ **do**
  $P_y \leftarrow ReduceExemplarSet(P_y, m)$
end for
for $y = s, ..., t$ **do**
  $P_y \leftarrow ConstructExemplarSet(X_y, m, \Theta)$
end for
$P \leftarrow (P_1, ..., P_t)$ # new exemplar sets

Algorithm 3 iCaRL - UpdateRepresentation

input $X^s, ..., X^t$ # training images of classes s, ..., t
require $P = (P_1, ..., P_t)$ # exemplar sets
require Θ # current model parameters

form combined training set
$D \leftarrow \bigcup_{y=s,...,t} \{(x, y) : x \in X^y\} \cup \bigcup_{y=1,...,s-1} \{(x, y) : x \in P^y\}$

# store network outputs with pre-update parameters
for $y = 1, ..., s-1$ **do**
  $q_i^y \leftarrow g_y(x_i)$ for all $(x_i, \cdot) \in D$
end for

run network training with loss function illustrated in equation 2.1

Algorithm 4 iCaRL - ReduceExemplarSet

input m # target number of exemplars
input $P = (p_1, ..., p_{|P|})$ # current exemplar set

$P \leftarrow (p_1, ..., p_m)$ # keep only the first m

output P # exemplar set

Algorithm 5 iCaRL - ConstructExemplarSet

input $X_y = \{x_1^y, ..., x_n^y\}$ # image set of class y
input m # target number of exemplars
require $\phi : X \rightarrow R^d$ # current feature map function

$\mu \leftarrow \frac{1}{n} \sum_{x \in X} \phi(x)$ # current class mean
for $k = 1, ..., m$ **do**
$p_k \operatorname*{argmin}_{x \in X} ||\mu - \frac{1}{k}[\phi(x) + \sum_{j=1}^{k-1} \phi(p_j)]||$
end for
$P \leftarrow (p_1, ..., p_m)$

output P # exemplar set

A.2 Experiment

iCaRL has been tested on the CIFAR-100 [18] dataset many times changing, for each trial, the number of classes per task. The used network is a ResNet32 [19] with the possibility to store up to 2000 exemplars. Here we have the results of the experiment.

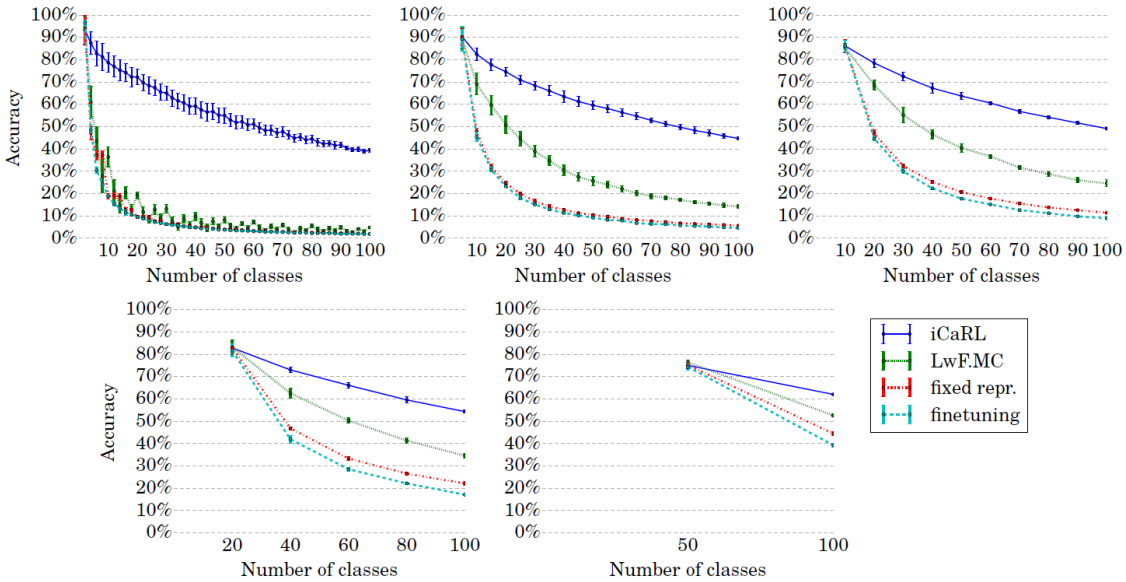

Figure A.1. Result (in blue) of *iCaRL* on the CIFAR-100 dataset.

Appendix B

End-to-End Incremental Learning

B.1 Pseudo code

The pseudo code of this method is very similar to the iCaRL's one, but some little differences:

- the classification is done traditional computing the logits and passing them through the softmax function;
- after the traditional training, there is an additional fine tuning phase using a balanced training set.

B.2 Experiment

End-to-End Incremental Learning has been tested on CIFAR-100 and ImageNet [20] datasets. For the first the used network is ResNet32 with the possibility to store up to 2000 exemplars. For the latter the used network is ResNet18 with the possibility to store up to 20000 exemplars. For both datasets many experiments have been executed changing, each time, the number of classes per task.
In figure B.1 there are two plots of the incremental accuracy on the CIFAR-100 using 2 and 5 classes per task.
In figure B.2 there is a summary of many experiments indicating the average incremental accuracy of this method on both the just introduced datasets and considering different values of classes per task.

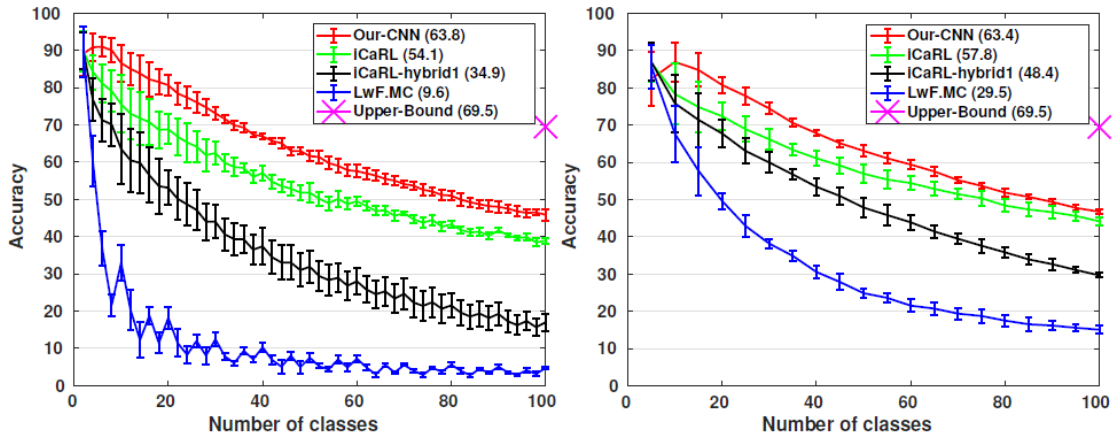

Figure B.1. Result (in red) of *End-to-End Incremental Learning* on the CIFAR-100 dataset with 2 and 5 classes per task.

# classes	2	5	10	20	50
Our-CNN	**63.8 ± 1.9**	**63.4 ± 1.6**	**63.6 ± 1.3**	**63.7 ± 1.1**	60.8 ± 0.3
iCaRL	54.1 ± 2.5	57.8 ± 2.6	60.5 ± 1.6	62.0 ± 1.2	**61.8 ± 0.4**
Hybrid1	34.9 ± 4.5	48.4 ± 2.6	55.8 ± 1.8	60.4 ± 1.0	60.8 ± 0.7
LwF.MC	9.6 ± 1.5	29.5 ± 2.2	40.4 ± 2.0	47.6 ± 1.5	52.9 ± 0.6

(a) CIFAR-100

# classes	10	100
Our-CNN	**90.4**	**69.4**
iCaRL	85.0	62.5
Hybrid1	83.5	46.1
LwF.MC	79.1	43.8

(b) ImageNet

Figure B.2. Average incremental accuracy of *End-to-End Incremental Learning* on the CIFAR-100 and ImageNet datasets with different values of classes per task.

Appendix C

Large Scale Incremental Learning

C.1 Pseudo code

The pseudo code of this method is very similar to the iCaRL's one, but the additional bias correction step.

Algorithm 6 Large Scale Incremental Learning - BIAS CORRECTION

input k # number of validation samples
require $P_{old} = (P_1, ..., P_s - 1)$ # stored samples
require $P_{new} = (P_s, ..., P_t)$ # new samples

$D_{val} \leftarrow ExtractValidationData(P_old, P_new, k)$ # validation data
freeze all layers but the Bias Correction one
run bias correction layer training on D_{val} data and loss 2.3

C.2 Experiment

Large Scale Incremental Learning has been tested on ImageNet-1000 and Celeb-10000 [21] datasets considering 100 and 1000 classes per task respectively. The used network is a ResNet18 for both datasets.
The figure C.1 illustrates the results of these experiments.

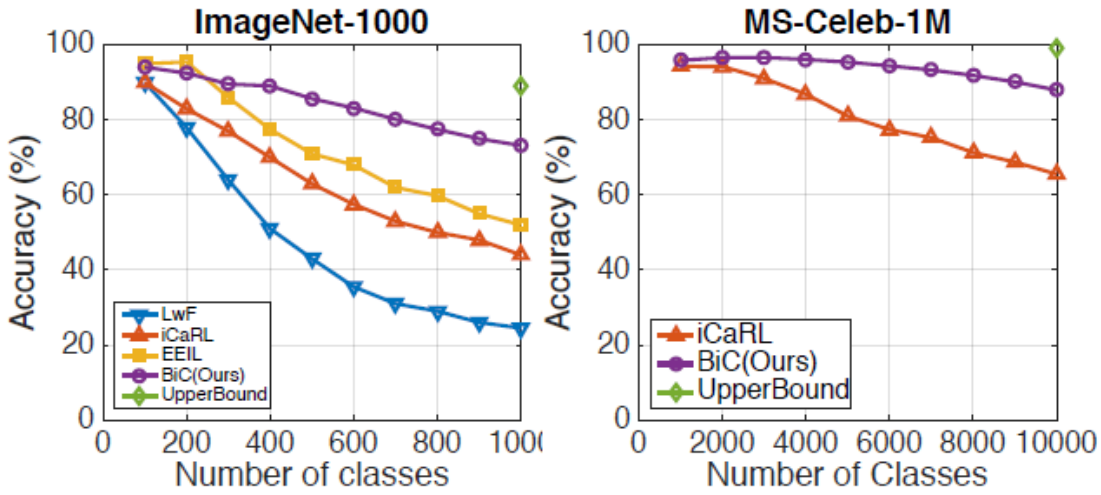

Figure C.1. Result (in purple) of *End-to-End Incremental Learning* on the Imagenet-1000 and Celeb-10000 datasets with 10 incremental steps.

Appendix D

Packnet

D.1 Pseudo code

Here we are going to illustrate the pseudo code for the training of a single task.
At inference time the procedure is quite simple. In fact, once combined the network's weights with the mask of the corresponding task, the classification works just like in the most general case.

Algorithm 7 Packnet - SINGLE TASK TRAINING

input k # percentage of weights to cut
input m # mask of available weights
require Θ # network weights
require D_t # data of the new task

$\Theta' \leftarrow MaskWeights(\Theta, m)$
$TrainNetwork(\Theta', D_t)$
$\Theta_{unrelevant}, \Theta_{relevant} \leftarrow SelectUnrelevantWeights(\Theta')$
PruneWeights($\Theta_{unrelevant}$)
$TrainNetwork(\Theta_{relevant}, D_t)$
$m \leftarrow UpdateMask(m, \Theta_{relevant})$ # reduce available weights

output $\Theta_{relevant}, m$

D.2 Experiment

Packnet: Adding Multiple Tasks to a Single Network by Iterative Pruning has been tested on ImageNet, Caltech-UCSD Birds [22], Standford Cars [23] and Oxford Flowers [24] datasets. Each dataset represents a classification task and, in the depicted example in figure 1.1, they have been considered just in this order. Instead of the accuracy, in that figure, the error percentage is indicated. The used network is a VGG16.

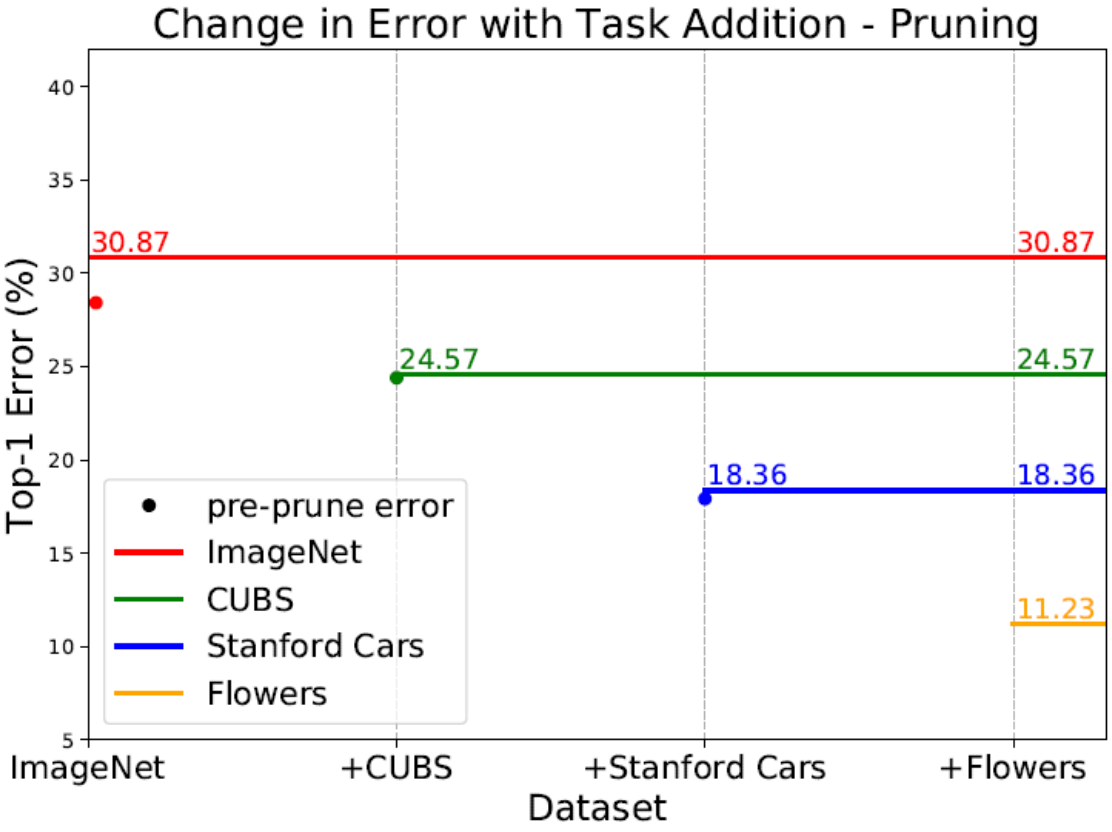

Figure D.1. Result of *Packnet* on the ImageNet, Standford Cars, Caltech-UCSD Birds and Oxford Flowers datasets.

Appendix E

Piggyback

E.1 Pseudo code

Here we are going to illustrate the pseudo code for the training of a single task.
At inference time the procedure is quite simple. In fact, once combined the network's weights with the mask of the corresponding task, the classification works just like in the most general case.

Algorithm 8 Piggyback - SINGLE TASK TRAINING

require Θ # network weights
require D_t # data of the new task

$m \leftarrow InitializeRealValuesMaskWeights()$
$TrainNetwork(\Theta', D_t, m)$
$m' \leftarrow Binarize(m)$

output m'

E.2 Experiment

Piggybackt: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights has been tested on ImageNet, CUBS, Standford Cars, Oxford Flowers, WikiArt [25], Sketch [26] datasets both in this and in the reverse order. Each dataset represents a classification task and, in the depicted example in figure E.1, instead of the accuracy, the error percentage is indicated. In

addition there is also the size of the model for each of the examined methods. The used network is a VGG16 [27].

Dataset	Classifier Only	PackNet [7] ↓	PackNet [7] ↑	Piggyback (ours)	Individual Networks
ImageNet	28.42 (9.61)	29.33 (9.99)		28.42 (9.61)	28.42 (9.61)
CUBS	36.49	22.30	29.69	20.99	21.30
Stanford Cars	54.66	15.81	21.66	11.87	12.49
Flowers	20.01	10.33	10.25	7.19	7.35
WikiArt	49.53	32.80	31.48	29.91	29.84
Sketch	58.53	28.62	24.88	22.70	23.54
# Models (Size)	1 (537 MB)	1 (587 MB)		1 (621 MB)	6 (3,222 MB)

Figure E.1. Result of *Piggyback* on the ImageNet, CUBS, Standford Cars, Oxford Flowers, WikiArt, Sketch datasets.

Appendix F

Exemplar-Supported Generative Reproduction

F.1 Pseudo code

Here we are going to illustrate the pseudo code for the training of a single class.
At inference time the procedure is the traditional one with the logits that are passed through the softmax activation function.

Algorithm 9 Exemplar-Supported Generative Reproduction - SINGLE CLASS TRAINING

input K # upper bound of stored exemplars
require $X^1_{real}, ..., X^{t-1}_{real}$ # stored exemplars
require $G_1, ..., G_{t-1}$ # generators of old exemplars
require X^t_{real} # new exemplars

$G_t \leftarrow TrainGenerator(X^t_{real})$ # objective function at 2.4
$X^1_{exem}, ..., X^{t-1}_{exem} \leftarrow GenSamples(G_1, ..., G_{t-1})$ # artificial old exemplars
$D_{train} \leftarrow X^t_{real} \cup X^1_{real}, ..., X^{t-1}_{real} \cup X^1_{exem}, ..., X^{t-1}_{exem}$
$TrainNetwork(D_{train})$
for $y = 1, ..., t$ **do**
    $X^y_{real} \leftarrow SelectExemplars(X^y_{real}, K)$
end for

output $X^1_{real}, ..., X^t_{real}$

Algorithm 10 Exemplar-Supported Generative Reproduction - SelectExemplars

input K
require X^y_{real}
require t # seen classes

$m \leftarrow K/t$ # exemplars per class
for $x^y_i \in X^y_{real}$ **do**
  $x^y_i, score_i \leftarrow Predict(x^y_i)$
end for
$x^y_1, ..., x^y_{|X^y_{real}|} \leftarrow SortByScore((x^y_1, score_1), ..., (x^y_{|X^y_{real}|}, score_{|X^y_{real}|}))$

output $x^y_1, ..., x^y_m$ # only the first m ones

F.2 Experiment

Exemplar-Supported Generative Reproduction for Class Incremental learning has been tested on CIFAR-100 and ImageNet-Dogs [28] datasets adding one class at a time. For the first a LeNet [29] and a WGAN [30] have been used as main network and generator respectively. For the latter a ResNet with 4 residual blocks and a AC-GAN [31] have been used as main network and generator respectively. In figure F.1 there are the results of these experiments.

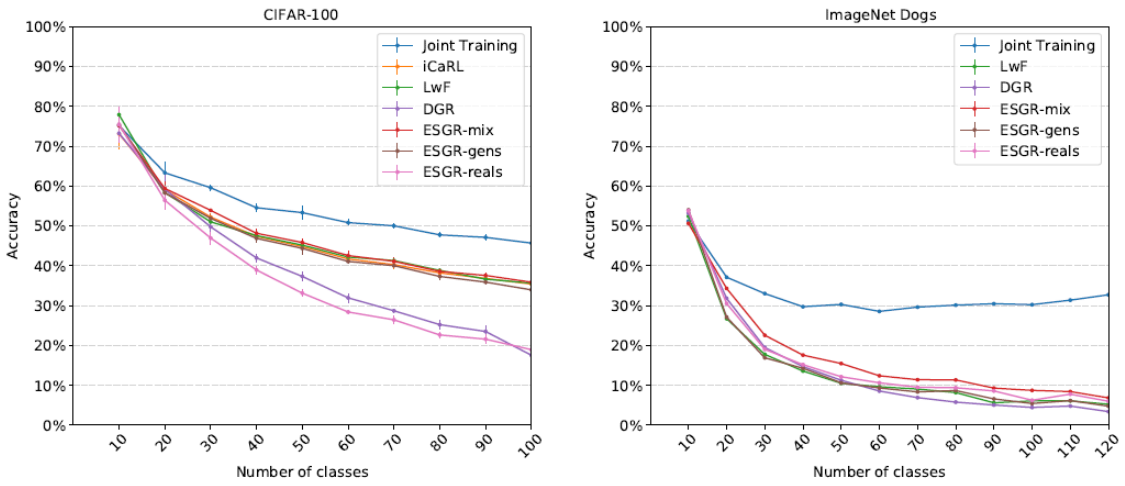

Figure F.1. Result (in brown) of *Exemplar-Supported Generative Reproduction* CIFAR-100 and ImageNet-Dogs datasets.

Appendix G

Deep Generative Replay

G.1 Pseudo code

Here we are going to illustrate the pseudo code for the training of a single task.
At inference time the procedure is the traditional one with the logits that are passed through the softmax activation function.

Algorithm 11 Deep Generative Replay - SINGLE TASK TRAINING

require G_{t-1}, S_{t-1} # last scholar (generator, solver)
require (X, Y) # new task samples

$X' \leftarrow GenerateOldSample(G_{t-1})$
$G_t \leftarrow TrainGenerator(X \cup X')$
$(X', Y') \leftarrow GenerateOldSampleWithLabel(G_{t-1}, S_{t-1})$
$S_t \leftarrow TrainSolver((X', Y') \cup (X, Y))$

output G_t, S_t

G.2 Experiment

Continual Learning with Deep Generative Replay has been tested on MNIST [32] and SVHN [33] datasets. These datasets have been presented one after the other to the model and in both the orders. In particular each task was made by a group of classes of a dataset. A WGAN-GP [34] netwotk has been used for generators.

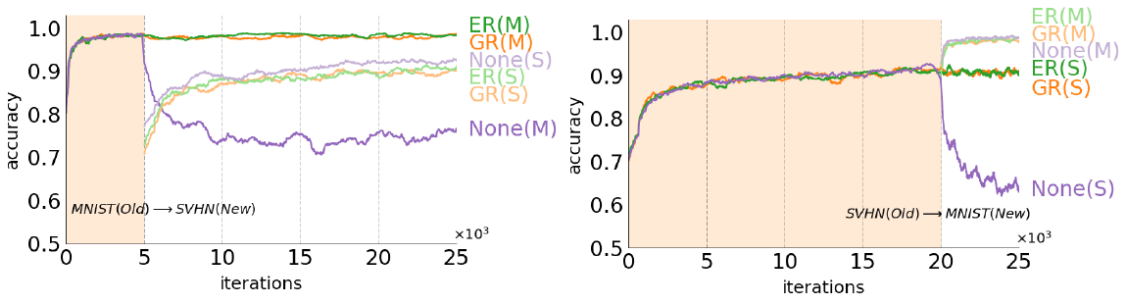

Figure G.1. Result (in orange) of *Deep Generative Replay* MNIST and SVHN datasets.

Appendix H

Expert Gate: Lifelong Learning with a Network of Experts

H.1 Pseudo Code

The training of autoencoders and experts, as explained into the dedicated sections, is quite simple. Here we are going to present the pseudo code for the inference procedure.

Algorithm 12 Expert Gate - INFERENCE PROCEDURE

input x # input sample
require $A_1, ..., A_t$ # trained autoencoders

for i=1,...,t **do**
    $p_i \leftarrow A_i(x)$
end for
$t^* \leftarrow \text{argmax}\, softmax(p_1, ..., p_t)$ # predicted task
$E_{t^*} \leftarrow LoadExpert(t^*)$
$y^* \leftarrow PredictClass(E_{t^*}, x)$ # predicted class

output y^*

H.2 Experiment

Expert Gate: Lifelong Learning with a Network of Experts has been tested on MIT Scenes [35], Caltech-UCSD Birds, Oxford Flowers, Stanford Cars, and FGVC Aircrafts [36] datasets. Each dataset represent a task to learn and they are presented to the model in the order they have been just presented. The used features extractor is an AlexNet [37] pretrained on ImageNet.
In figure H.1 there are the results of incremental classification, while in figure H.2 there are the results of the gate analysis.

Method	Scenes	Birds	Flowers	Cars	Aircrafts	Actions	avg
Joint Training*	59.5	56.0	85.2	77.4	73.4	47.6	66.5
Most confident model	40.4	43.0	69.2	78.2	54.2	8.2	48.7
Expert Gate	60.4	57.0	84.4	80.3	72.2	49.5	67.3

Figure H.1. Incremental accuracy of *Expert Gate: Lifelong Learning with a Network of Experts.*

Method	Scenes	Birds	Flowers	Cars	Aircrafts	Actions	avg
Discriminative Task Classifier - *using all the tasks data*	97.0	98.6	97.9	99.3	98.8	95.5	97.8
Expert Gate (ours) - *no access to the previous tasks data*	94.6	97.9	98.6	99.3	97.6	98.1	97.6

Figure H.2. Gate analysis of *Expert Gate: Lifelong Learning with a Network of Experts.*

Bibliography

[1] S. Rebuffi, A. Koleskinov, G. Sperl, C. Lampert. *iCaRL: Incremental Classifier and Representation Learning*, CVPR2017, arXiv:1611.07725.

[2] F. M. Castro, M. J. Marin-Jiménez, N. Guil, C. Schmid, K. Alahari. *End-to-End Incremental Learning*, ECCV2018, arXiv:1807.09536.

[3] Y. Wu, Y. Chen, L. Wang, Y. Ye, Z. Liu, Y. Guo, Y. Fu. *Large Scale Incremental Learning*, CVPR 2019, arXiv:1905.13260.

[4] A. Mallya, S. Lazebnik. *Packnet: Adding Multiple Tasks to a Single Network by Iterative Pruning*, CVPR2018, arXiv:1711.05769.

[5] A. Mallya, D. Davis, S. Lazebnik. *Piggybackt: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights*, ECCV2018, arXiv:1801.06519.

[6] M. Courbariaux, Y.Bengio, JP. David. *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*, NIPS 2015, arXiv:1511.00363.

[7] M. Courbariaux, I. Hubara, R. El-Yaniv, Y.Bengio. *Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*, NIPS 2016, arXiv:1602.02830.

[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio. *Generative adversarial nets. In Advances in Neural Information Processing Systems*, NIPS 2014, arXiv:1406.2661.

[9] C. He, R.Wang, S. Shan, X. Chen *Exemplar-Supported Generative Reproduction for Class Incremental learning*, BMVC2018.

[10] H. Shin, J. K. Lee, J. Kim, J. Kim *Continual Learning with Deep Generative Re-play*, NIPS 2017, arXiv:1705.08690.

[11] R. Aljundi, P. Chakravarty, T. Tuytelaars. *Expert Gate: Lifelong Learning with a Network of Experts*, CVPR 2017, arXiv:1611.06194.

[12] H. Bourlard, Y. Kamp. *Auto-association by multilayer perceptrons and singular value decomposition*, Biological cybernetics, 59(4-5):291–294, 1988.

[13] I. Goodfellow, Y. Bengio, A. Courville. *Deep learning*, Book in preparation for MIT Press.

[14] G. E. Hinton, R. R. Salakhutdinov. *Reducing the dimensionality of data with neural networks*, Science, 313(5786):504–507, 2006

[15] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, et al. *On rectified linear units for speech processing*, IEEE International Conference on Acoustics, Speech and Signal Processing, pages 3517–3521, 2013.

[16] G. Hinton, O. Vinyals, J. Dean. *Distilling the knowledge in a neural network*, NIPS 2015, arXiv:1503.02531.

[17] Z. Li, D. Hoiem. *Learning without forgetting*, European Conference on Computer Vision, pages 614–629, Springer, 2016.

[18] A. Krizhevsky *Learning Multiple Layers of Features from Tiny Images*, 2009.

[19] K. He, X. Zhang, S. Ren, J. Sun. *Deep Residual Learning for Image Recognition*, arXiv:1512.03385.

[20] J. Deng, W. Dong, R. Socher, L. Li, K. Li and L. Fei-Fei *ImageNet: A Large-Scale Hierarchical Image Database*, CVPR09, 2009.

[21] Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang, Xiaoou. *Deep Learning Face Attributes in the Wild*, ICCV15, 2015.

[22] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, P. Perona. *Caltech-UCSD Birds 200*, California Institute of Technology, 2010.

[23] J. Krause, M. Stark, J. Deng, L. Fei-Fei. *3D Object Representations for Fine-Grained Categorization*, ICCV 2013.

[24] M-E. Nilsback, A. Zisserman. *Automated flower classification over a large number of classes*, Indian Conference on Computer Vision, Graphics and Image Processing, 2008.

[25] *WikiArt dataset* available at https://www.wikiart.org/.

[26] P. Sangkloy, N. Burnell, C. Ham, J. Hays. *The Sketchy Database: Learning to Retrieve Badly Drawn Bunnies*, ACM Transactions on Graphics (proceedings of SIGGRAPH), 2016.

[27] K. Simonyan, A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv:1409.1556v6.

[28] A. Khosla, N. Jayadevaprakash, B. Yao, L. Fei-Fei. *Novel dataset for Fine-Grained Image Categorization*, CVPR 2011.

[29] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. *Gradient-based learning applied to document recognition* Proceedings of the IEEE, 86(11):2278–2324, 1998.

[30] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, A. C. Courville.

Improved training of wasserstein gans, In Advances in Neural Information Processing Systems, pages 5769–5779, 2017.

[31] A. Odena, C. Olah, J. Shlens. *Conditional image synthesis with auxiliary classifier gans*, 2016, arXiv:1610.09585.

[32] Y. LeCun, C. Cortes. *MNIST handwritten digit database*, http://yann.lecun.com/exdb/mnist/, 2010.

[33] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A. Y. Ng. *Reading Digits in Natural Images with Unsupervised Feature Learning*, NIPS, 2011.

[34] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, A. Courville. *Improved training of wasserstein gans*, 2017, arXiv:1704.00028.

[35] A. Quattoni, A.Torralba. *Recognizing Indoor Scenes*, CVPR 2009.

[36] S. Maji, J. Kannala, E. Rahtu, M. Blaschko, A. Vedaldi. *Fine-Grained Visual Classification of Aircraft*, 2013, arXiv:1306.5151.

[37] A. Krizhevsky, I. Sutskever, G. E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS 2012.

[38] O. Linda, T. Vollmer, M. Manic. *Neural network based intrusion detection system for critical infrastructures*, IJCNN 2009.

[39] Y. Gal. *Uncertainty in Deep Learning*, PhD thesis, University of Cambridge, 2016.

[40] Y. Kwon, J-H. Won, B. J. Kim, M. C. Paik *Uncertainty quantification using Bayesian neural networks in classification: Application to ischemic stroke lesion segmentation*, 2018

[41] D. JC. MacKay. *A practical bayesian framework for backpropagation networks*, Neural computation, 4(3):448–472, 1992.

[42] R. M. Neal. *Bayesian learning via stochastic dynamics*, Advances in neural information processing systems, pages 475–482, 1993.

[43] C. Louizos, M, Welling.*Multiplicative normalizing flows for variational bayesian neural networks*, International Conference on Machine Learning, pages 2218–2227, 2017a.

[44] C. Louizos, M, Welling.*Multiplicative normalizing flows for variational Bayesian neural networks*, Proceedings of the 34th International Conference on Machine Learning, pages 2218–2227, 2017b.

[45] A. Kendall, Y. Gal. *What uncertainties do we need in bayesian deep learning for computer vision?*, Advances in Neural Information Processing Systems, pages 5580–5590, 2017.

[46] C. Szegedy, S. Ioffe, V. Vanhoucke, Alex Alemi. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*, CVPR 2016.

[47] https://en.wikipedia.org/wiki/Otsu