SAPIENZA
UNIVERSITÀ DI ROMA

Techniques for improving continual learning in deep networks

Ingegneria dell'informazione, informatica e statistica-

Corso di Laurea Magistrale in Artificial Intelligence and Robotics

Candidate

Jary Pomponi
ID number 1566973

Thesis Advisor

Aurelio Uncini

Co-Advisor

Simone Scardapane

Academic Year 2018/2019

Techniques for improving continual learning in deep networks
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: jarypomponi@gmail.com

Grazie

Abstract

write your abstract here

Contents

1 Introduction **1**

2 Artificial neural networks (ANN) **2**
2.1 Introduction 2
2.2 Network topology 2
2.2.1 Multi layer perceptron (MLP) 2
2.2.2 Convolutional neural network (CNN) 3
2.3 Activations functions 4
2.3.1 Fixed activations functions 4
2.3.2 Fixed and parametric 4
2.3.3 Parametric activation functions 5
2.4 Train the network 5

3 Kaf nets **7**
3.1 Introduction and general overview 7
3.2 Kaf 7
3.3 Multikaf 8

4 Continual Lifelong Learning in ANNs **10**
4.1 Introduction 10
4.1.1 Continual Learning scenarios 10
4.1.2 Regularization approaches 11
4.1.3 Structural approaches 13
4.1.4 Rehearsal approaches 13

5 Proposed approach: embedding regularization **15**

6 Experiments **17**
6.1 MINST 17
6.2 CIFAR10 20
6.3 CIFAR100 20

Bibliography **21**

Chapter 1

Introduction

In this thesis ...

Chapter 2

Artificial neural networks (ANN)

2.1 Introduction

An artificial neural network (ANN) is a function $y = f(x; \theta)$. This function has a particular structure and the value θ is the core of it. The popularity of this kind of function came from the universal approximation theorem, it states that a neural network can represent a wide variety of functions, given the appropriate parameters θ. The fist version of the approximation theorem was proven in [3] for a network with sigmoid activation function (the activation functions will be exposed later), while in [9] the authors showed that is not a specific choice of the activation function that matters, but rather the architecture of the network.

In the next sections we will see all the base components that define an ANN:

- Structure, or topology
- Neuron response characteristic, the non linearity function present in neurons
- A learning procedure

2.2 Network topology

2.2.1 Multi layer perceptron (MLP)

The base component of an ANN is called artificial neuron. It is a function f_i of the input $x \in R^n$ weighted by a vector $w_i \in R^n$ plus a term called b_i bias. Without some non-linearity the function will be only a linear combination of the inputs, for this reason a function ϕ is applied to the result: $y_i = f_i(x; w_i) = \phi(< x, w_i > +b_i)$. The activation functions are one of the main topic covered by this thesis, and a wide variety of them will be shown later. Combining multiple neurons together creates a layer; in this case the output will be $y_l = (y_1, \dots y_m)$ where m is the size of the layer. A network with a single layer is called perceptron.

Combining multiple layer together create a multi-layer perceptron, where the first layer takes as input the x and a layer l_i inside the network takes y_{i-1} as input. This architecture is very basic because the neuron in the same layer are not connected. Usually the activation functions used in the last layer is different from the ones used in the others layers, this because the desired output depends on the problem that the network is built for. Mathematically, the multi-layer perceptron can be written as:

for the hidden layers $i = 1...L$

$$a_i(x) = b_i + w_i h_{i-1}(x) \tag{2.1}$$
$$h_i(x) = \phi_i(a_i(x)) \tag{2.2}$$

with the output layer:

$$a_{L+1}(x) = b_{L+1} + w_{L+1} h_L(x)$$
$$h_L(x) = \phi_{L+1}(a_{L+1}(x)) = f(x;\theta)$$

where $f(x;\theta)$ is the function defined at the begin of this chapter. In our case Θ is the combination of all the weights and biases, respectively w_i and b_i with $i = 1 \dots L+1$. Find the correct set of parameters θ is the challenge and the key of the neural networks and it is done with a process called train.

2.2.2 Convolutional neural network (CNN)

For some types of data, like images, the MLP is not well suited, since it works with vectors. Hence, to train the MLP on images these should be converted to vectors, loosing the spatial information contained in the images. The CNN, introduced in [12], are able to process images and extract spatial information from adjacent pixels; actualy the cnns can process any matrix, or tensor, like input. Now the cnns are the standard when dealing with images.

The cnns are composed convolutional layers; each layers are composed by kernels that extract information from the pixels and weights these. In the case of gray-scale images we sill ave 2D images, so the kernels are 2d, since we process height and width, and the formula for the convolution between two function are:

$$(K * I)(i,j) = \sum_{m,n} K(m,n) I(i+n, j+m)$$

where K is the 2D kernel and I is the image, also 2D. The principle of this layer is to move the convolutional kernel on the image and at each step extract the convolution between the kernel and the part of the image that is currently treated; the kernel movement, on a pixel basis, is governed by a parameter s called stride. a lot of other stuffs are involved in a cnn; for example the image can be zero padded, in order to control the size of the output. The kernel, also called filter, is usually a square $k \times k$, and the result when applied on a portion of the image x, also big $k \times k$, is $w^t x + b$ where w are the weights of the filter, b the bias, and x are the pixels in the region of the image; in this layer the parameters are w and b. Multiple filter can be used, in this case the result will be stack of matrices. Also the output size is important: if the input image has $width_i \times height_i \times channels_i$, where $channels$ are 3 for RGB image and 1 for gray-scale, the size of the output matrix will be $width_o \times height_o \times filters$ with:

$$width_o = \frac{width_i - k + 2p}{s} + 1$$
$$height_o = \frac{height_i - k + 2p}{s} + 1$$

where p is the size of the padding and $filters$ are the number of filters used. As always, the convolutional operation are combined with an an activation function ϕ, usually the ReLU.

Another important layer used in the cnn are called pooling layer. The purpose if this kind of layer is to take the representation and makes it smaller and more manageable. It also has a kernel, but this time the operation depends on the kind of pooling, for example can be max pooling, where only the max values in the portion of the image are returned, or average pooling, where the output will be the average value of the pixels in the processed region. Should be noticed that this kind of layer has no parameters.

2.3 Activations functions

2.3.1 Fixed activations functions

The activation functions are crucial in every ANN. First of all they are the key of the ability of the networks to approximate every functions; without these non linear functions the output of the network can be reduced to a simply linear combination and stacking multiple layer will be useless. Also the activation functions are crucial in train phase of the network.

In this section we will show some functions.

2.3.2 Fixed and parametric

We briefly show some common fixed activation functions for ANN, that are the basis for the parametric ones. Back in the time before deep model, most of the functions were of squashing type; means that they where monotonically non-decreasing functions satifying:

$$\lim_{x \to -\infty} g(x) = c, \quad \lim_{x \to \infty} g(x) = 1,$$

where c can be 0, sigmoid function, or -1, hyperbolic tangent function. In practice squashing functions were not good for deep architecture, being prone to vanishing gradient, when the gradients tends to zero and the networks cannot learns, or exploding gradient, the opposite of vanishing gradient with the gradients that become too large; this due to the bounded derivatives ([**?**]).

One of the most popular and widely used activation function in deep models is called rectifier linear unit (ReLU) function, defined as:

$$g(x) = max\{0, x\}$$

This function has proven to be extremely efficient ([4], [18]), despite the facts that is unbounded and introduced a point of non differentiability. But ReLU bring some advantages: first, its gradient its either 0 or 1, making back-propagation efficient; secondly, the output its sparse. A smoothed version of the ReLU, called softplus, where introduced in [**?**] and is defined as:

$$g(x) = log(1 + e^x)$$

The gradient property of ReLU could be a problem if the weights are not good, since the function can get stuck at 0. This problem is called dying ReLU. To overcome this problem in [18] a new ReLU function called leaky ReLU where introduced:

$$g(x) = \begin{cases} x \quad if \quad x > 0 \\ \alpha x \quad otherwise \end{cases}$$

with $\alpha > 0$ generally set to a small value. With this function we lose the sparsity property but the network should be capable of escaping from bad regions improving the training.

Another "problem" problem of activation functions having only non negative output value is that the mean will always be positive. To circumnavigate this issue [2] introduced the Exponential linear Unit (ELU):

$$g(x) = \begin{cases} x \quad if \quad x > 0 \\ \alpha(e^x - 1) \quad otherwise \end{cases}$$

In this case the ELU modify the ReLU such that the negative part saturates at a user defined value α.

2.3.3 Parametric activation functions

In the previously introduced functions we had some parameters, such as α, that the user should set. One way to increase the flexibility of a neural network is to allow the network to learn these parameters, as long as functions remain differentiable. For example, in [6] a parametric version of the leaky ReLU (called PReLU) were introduced, where α is initialized at 0.25 and adapted for each neuron independently; the derivative of PReLU is very simple since is 0 if the input is greater or equal then 0, the input otherwise.

In [24] a parametric version of ELU (PELU), with an additional scalar parameter, were introduced as:

$$g(x) = \begin{cases} \frac{\alpha}{\beta}x \quad if \quad x > 0 \\ \alpha(e^{\frac{x}{\beta}} - 1) \quad otherwise \end{cases}$$

where both α and β are randomly initialized and adapted during the train phase.

Should be noted that, in case of l_p regularization the parameters present in PReLU should not be regularized, since it would bias the optimization process towards classical activation functions; on the contrary the parameters α and β in PELU should be regularized to prevent a degenerate behavior in which the linear weights are small and the parameters large.

2.4 Train the network

Exists many different ways to train a network and usually the choice depends on the available dataset. In this thesis only the supervised approach will be used, means that our dataset will have both x and y, where y is the label associated to x. Some others methods are unsupervised learning or reinforcement learning.

The fist step in order to train a network, and find a good set of parameters, is to choose a loss function. Loss function $L(\dot{})$ should measure, given a sample x, the difference between the real label y and the output of the network $\bar{y}$; obliviously the choice of this function depends on the problem. The expected loss is:

$$L(\theta) = \mathbf{E}_{(X,Y)\sim D}[l(f(X|\theta), Y]$$

where l is the chosen loss function but, in order to estimate θ, the empirical loss is used:

$$\bar{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(f(X_i|\theta), Y_i]$$

given a train set $\{(X_i, Y_i)\}_{i=1\ldots N}$. At each step the gradient w.r.t. loss is computed and back-propagated thought the network and the parameters are changed accordingly.

To know how much a weight affected the error the partial derivative is calculated:

$$\frac{\partial E}{\partial \theta_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial \theta_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial \theta_{ij}}$$

noticing that in $\frac{\partial a_j}{\partial \theta_{ij}}$ only one term depends in θ_{ij} we have $\frac{\partial a_j}{\partial \theta_{ij}} = h_i$; if the neuron is the first after the input one then it is only x_i. The derivative of the output of the neuron is $\frac{\partial h_j}{\partial a_j} = \frac{\partial \phi_j(a_j)}{\partial a_j}$ and depends on the activation function used. Follow that the activation function needs to be differentiable.

In the end the first term is $\frac{\partial E}{\partial h_j} = \frac{\partial E}{\partial y} = \frac{L(x,y)}{\partial \phi(y)}\frac{\partial \phi(y)}{dy}$ if the layer is the last one, otherwise the derivative is less easier. The final results is:

$$\frac{\partial E}{\partial \theta_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial \theta_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial a_j} h_i = h_i \delta_j$$

with

$$\delta_j = \begin{cases} \frac{\partial L(a_j, y)}{\partial}\frac{dh_j}{da_j} & \text{if j is an output neuron} \\ (\sum_{i=1}^{L} w_{jl}\delta_l)\frac{dh_j}{da_j} & \text{if j is an inner neuron} \end{cases}$$

The way to use these values depends on the optimizer used to train the network. The simplest one is called gradient descend and in this case we have $\Delta w_{ij} = -\eta h_i \delta_j$, with η a parameter chosen by the user and called learning rate. This parameter is very important since the size of the updates depends on it. A small value leads to a very slow training phase but a too high leads to a divergence of the training.

Chapter 3

Kaf nets

3.1 Introduction and general overview

Property of the European Southern Observatory...

3.2 Kaf

The kaf nets are a class of ANN that uses kernel function as activation function. Each activation function is modeled in terms of a kernel expansion over a dictionary D:

$$g(x) = \sum_{i=1}^{D} \alpha_i \kappa(x, d_i)$$

where the α_i is called mixing coefficient, d_i is called dictionary element and $\kappa(\cdot,\cdot)$ $R \times R \to R$ if a 1 dimension kernel function [8]. Usually, in kernel methods, the dictionary is selected from the training data, but this means that D would grow linearly, unless some strategy are implemented. To simplify the activation functions, the dictionary contains a fixed numbers of values, that can be trained or not, and the mixing coefficients are learned based on these values. Different methods can be used to choose the values in the dictionary; the most straightforward, and the one used in this thesis, consist in pick n values from a range [-B, B], uniformly around the zero. This approach is not new, and these is a vast literature about kernel functions using fixed dictionary elements [24]

The kernel function $\kappa(\cdot,\cdot)$ needs to respect the semi-definiteness property; for any possible choice of the mixing coefficients and the dictionary elements holds:

$$\sum_{i=1}^{D} \sum_{j=1}^{D} \alpha_i \alpha_j \kappa(d_i, d_j) \geq 0$$

Different kernels functions can be used, for example:

- Gaussian: $\kappa(x, y) = exp\{-\gamma(x - y)^2\}$
- Relu:
- Polynomial: $\kappa(x, y) = (c + xy)^p$

- Elu:

Kaf activation functions are equivalent to learning linear functions over a large number of nonlinear transformations of the original layer, without compute these transformations. More than that the Gaussian kernel has a local property, means that the the mixing coefficients have only a local effect on the shape of the activation function.

The shape of the kaf activation function depends on its parameters. If the dictionary is fixed than the shape depends only on the mixing coefficients. In fact the initialization of these are crucial in order to allow proper training of the network; draw these coefficients from a normal distribution provides good diversity. Another advantage of this approach is that the coefficients can be initialized so that a certain behavior is obtained; for example an activation function can be replicated.

More in deep, having a vector $\boldsymbol{t}$ of desired points points, one for each element of the dictionary $\boldsymbol{d}$, the mixing coefficients can be initialized using kernel ridge regression:

$$\boldsymbol{\alpha} = (\boldsymbol{K} + \epsilon \boldsymbol{I})^{-1} \boldsymbol{t}$$

where $\boldsymbol{K} \in R^{D \times D}$ is the kernel matrix computed between $\boldsymbol{t}$ and $\boldsymbol{d}$, and ϵ is used to avoid degenerate solutions.

The Gaussian kernel has another parameter: the kernel bandwidth. This parameter is crucial in order to achieve a good behaviour, since it acts indirectly on the effective number of adaptable parameters. In the literature many methods have been proposed; in the kaf nets, instead of leaving the γ as another hyper-parameter, the following rule of thumb is used:

$$\gamma = \frac{1}{6\delta^2}$$

where δ is the distance between any two points in the dictionary.

3.3 Multikaf

Assume to have a set of kernel functions $M = \kappa_i \dots \kappa_m$, there is a vast field of research on how successfully combine the kernels to obtain a new one; this method is called Multiple Kernel Learning [1]. Each Multi-kaf activation function use all the kernels in the following way:

$$f(x) = \sum_{i=1}^{D} \alpha_i \sum_{j=1}^{M} \mu_j \kappa_j(x, d_i)$$

where $\boldsymbol{\mu} = \mu_{j=1M}$ is another set of trainable parameters that weights the importance of the kernels. Should be noted that this method introduces only M-1 new trainable parameters for each neurons, and, if the dictionary is fixed and non adaptable, all the kernels are evaluated on the same points, making the simpler and faster.

Also some other methods to combine different kernels into one have been used:

- softmax: the μs in the above equation are replaced by a distribution build using a softmax, so that $\sum_i \mu_i = 1$
- sigmoid: in this case each μ value is passed thought a sigmoid function, shrinking the value in 0 and 1.
- layer attention: in this method $\boldsymbol{\mu}$ is generated each time the kernel function is called, using a function that uses the input of the layer: $\boldsymbol{\mu} = f(x) = softmax(Mx + b)$, where W is a matrix and b a vector; these values are adapted during the train phase. This will produce m values shared across all the neurons in the layer.
- neurons attention: the idea is the same used in layer attention method but, in this case, the set $\boldsymbol{\mu}$ is estimated for each neuron.

All the method above have been used and the results will be shown later.

Chapter 4

Continual Lifelong Learning in ANNs

4.1 Introduction

Lifelong learning represent a challenge for machine learning methods, this is due the effect called catastrophic forgetting. This event is the tendency of the machine learning model to forget learned knowledge when learning novel observations [25]. A lifelong learning system should be capable of learn from a continuous stream of information, without forget the past. Usually these information are splitted into tasks and the number of tasks are not predefined. So, in general, the model should be capable of learning and correctly classify new tasks without forget the previously learned ones.

The most straightforward system consists in a memory system that stores old information, so the network is learned on new data and a bunch of old samples drawn from the memory [20, 21]. The principal drawback of these methods is that the memory required by the system grown at each task, since all the samples should be saved. More sophisticated methods have been developed, and in the next section some of them will be introduced and compared. Conceptually these approaches in resolving catastrophic forgetting can be divided in three sets: Regularization Approaches, Dynamic architectures and Complementary Learning Systems and Memory Replay.

In this thesis I will analyze in only some of the existing techniques, but an overview of the others will be given as well.

4.1.1 Continual Learning scenarios

The simplest scenario is called data permutation and each tasks is simply a permutation, random or not, of the input features vectors. An identical permutation is also applied on the test data. In this case the output size of the network is fixed and the tasks overlaps, and each training on a task contains the same information and classes of the others.

The mostly studied scenarios is called Multi-Task (MT), where the model should learns incremental number of isolated task without forget the old ones. The accuracy is computed for each task separately. A model trained in this scenario cannot be used to classify an unknown sample across all the task, since the network has been trained imposing the isolation of the tasks. This scenario are suited for studying problems in which the tasks can be separated completely but not for an incremental scenario. Despite this this is the most used scenario, and is the one that I will consider for the experiments.

A new and unexplored scenario is denoted as Single Incremental Task and (SIT) is addressed in [19]. An example of SIT is called class-incremental learning where the classes are added sequentially, as in the MT scenario, but these are not separated and the classification is calculated on all the task encountered so far. Train a network is capable of classify unknown image among all the classes encountered so far, but is also harder to train since it still have to deal with catastrophic forgetting and also should be capable of discriminate and classify correctly classes that are never seen together during the train.

Another scenario is called Multi-Modal learning and the model incrementally learns different dataset; for example image classification and then audio classification. The goal is to determine if a model can learn tasks that are completely different not only in the number of classes but also in the features given as input to the neural network. A model that is capable of this will be more efficient than building separated models for each modality.

4.1.2 Regularization approaches

The main idea at the base of these approaches consists in alleviate the catastrophic forgetting by imposing some kind of constraints on training the new tasks.

In [14] the authors proposed a method called Learning without forgetting (LwF); it is composed of convolutional neural networks in which a network trained on an old task is enforced to be similar to the network of the new task by using a technique called knowledge distillation ([7]). In this method we have a set of parameters Θ_s shared across all the tasks and a set of parameters Θ_{new}. These parameters are optimized at the same time, imposing additional constraints such that the prediction on the new samples using Θ_s and the parameters of old tasks Θ_o wont be shifted too much, in order to remember how to classify correctly old tasks. Given the train data (X_n, y_x) of a new task, the output of past task y_o and a new parameter Θ_n, the update for the parameters are given by:

$$\Theta_s^*, \Theta_n^*, \Theta_o^* \leftarrow \arg \max_{\Theta_s \Theta_n \Theta_o} L_o(y_o, \tilde{y}_o) + L_n(y_n, \tilde{y}_n) + R(\Theta_s\, \Theta_n\, \Theta_o)$$

where L_o and L_n minimizes the difference between the predicted task and the real one; in the first case the task is an old one and in the second one the task is the current. Also R is a regularization term over the parameters. The main drawbacks are the required training time, that increases with the number of tasks, and the fact that the loss depends on how an old task is relevant. Additionally the distillation technique requires to store data for each task.

The Elastic Weight Consolidation is another technique, proposed in [11], for continual learning and supervised scenarios. The idea is to penalize, and slow down, the change of the parameters that are important for old tasks while learning the new one. The importance of the parameters Θ with respect to an old task T_o is modelled as a posterior probability $p(\Theta, T)$. Given a task T_a and another independent task T_b, the log value of the posterior probability is:

$$\log p(\Theta|D) = \log p(D_b|\Theta) + \log p(\Theta|D_a) - \log p(D_b)$$

This method approximate $\log p(\Theta|D_a)$, which embeds the information about the previous task, as a Gaussian distribution with mean Θ_a^*, the parameters at the end of task D_a, and diagonal variance given by the fisher information matrix F. In the end the loss functions is:

$$\log p(\Theta) = L_b + \sum_i \frac{\lambda}{2}(F_i(\theta_i - \theta^*_{A,i})^2$$

E where L_b denotes the loss on the current task b, λ is a term used to set the relevance of the old tasks, and i denotes the index of the parameter in Θ. The matrix F, of a true distribution $p(y|x;\Theta)$, is an empirical, and biased, version of the real one and it is calculated as follow:

$$F = \mathbf{E}_{x\sim\phi}\left\{ \mathbf{E}_{y\sim p(y|x;\Theta)} \left[\frac{\partial^2 \log p}{\partial \Theta^2} \right] \right\}$$

where ϕ is the empirical distribution of the training set. The second derivative is important to understand how the information matrix can prevent the forgetting of old tasks. Once the train phase is over, the information matrix indicates how prone each parameter is to causing forgetting; it is referable to move along direction with low F, since $logp$ decreases slowly. In the end, EWC use F to to penalize moving in directions where the Fischer value are higher since can lead to forget past tasks.

This method performs well in a non incremental scenarios, as shown in [10] and in the experiments that will be shown. A better version of EWC, focused on incremental scenario, can be found in [15].

Should be noted that EWC requires that the information about the old tasks are stored; this can be avoided if we think that the new network is a network that contains the information about the past tasks and, since the training regularize the parameters, is not needed to save all the past tasks data, but only the last one. Also the information matrix is updated online as new tasks are observed: $F = F + F_i$. This version is called online EWC and is faster and requires less memory but leads to same results of the offline EWC.

A variant of EWC was introduced in [27], is called Synaptic Intelligence and the main difference is that the parameters importance is calculated online during the train. Given the update step during the train on the task t:

$$\delta L_i = \Delta\theta_i \cdot \frac{\partial L}{\partial \theta_i}$$

where $\delta\theta_i$ is the update of parameter i and $\frac{\partial L}{\partial \theta_i}$ the gradient. The total loss associated to a single parameter is the sum of all the $\sum \Delta L_i$ calculated during the training. The importance associated to a parameter i is given by:

$$F_i = \frac{\sum \Delta L_i}{T_k^2 + \epsilon}$$

where T_k is the total movement of the parameters θ_k during the training of the task and ϵ is a constant to avoid division by zero.

In this section some approaches that change the network structure when new information become available by changing, for example the size of the layers. The approaches

4.1.3 Structural approaches

The structural approaches alleviate the catastrophic forgetting by modifying the structure of the network when is necessary. In this section only some methods will be exposed, since the structural approaches are not the main focus of this thesis.

One of the most popular method is called Progressive Network and was introduced in [22]. When a new task is encountered the parameters associated to the old tasks are blocked, preventing any further changes, and a new neural network is created and the existing connection with the lateral connections with the old tasks are learned. To avoid catastrophic forgetting only the new connections will be optimized. The experiments reported a good results, outperforming common baseline approaches. Intuitively, the main drawback is that the complexity of the network increase with the number of learned task.

A very simple approach that ave showed good results in a SIT scenario is called Copy Weight with Reinit (CWR) and was introduced in [16]. The base idea is that the most straightforward way to implement a SIT strategy is to freeze the parameters after each task and, when a new task is encountered, extend the output layer by copying the optimal old output weights into the new one. The old weights could be frozen or fine tuned. To learn the new weights CWR keeps two sets of weights for the output classification layer: *cw* are the consolidated weights used for inference and *tw* are the weights used in the train phase; *cw* are randomly initialized before the first task and *tw* are randomly initialized before each new task. At the end of each task *tw* are scaled and copied in *cw*.

In [23] a combination of self-organizing incremental neural network and a pre-trained CNN network was proposed in order to use the representation power given by CNN and allow the network to grow and adapt in a continual learning scenario. Again, the main problem of this approach is the scalability, since the required memory grow with the number of task. Another problem is that relying on a pre-trainded network affect the power of the network, and it depends on the dataset used to train the pre-trained network.

4.1.4 Rehearsal approaches

In this section will be presented methods that use past information to strengthen the connections in the network that are important for past tasks. The most straightforward method consist in storing part of past training tasks and feed into the network both new and old data while training new tasks.

One of the most interesting rehearsal methods is called Gradient Episodic Memory and was proposed in [17]. In GEM the past information are used to constraints current gradients to avoid the increase of losses associated to past tasks but allowing their decrease; this could lead to positive transfer of knowledge to past tasks. The method alleviate catastrophic forgetting if one of the following constraints are violated:

$$< g, g_i >= \left\langle \frac{\partial L(y|x,\Theta)}{\partial \Theta}, \frac{\partial L(y_i|x_i,\Theta)}{\partial \Theta} \right\rangle \quad \text{for all past tasks i}$$

To do this a Dual Quadratic Program with inequality constrains are solved:

$$\begin{aligned} &\text{minimize}_v && \frac{1}{2}v^T G G^T v + g^T G^T v \\ &\textit{subject to} && v \geq 0 \end{aligned}$$

where g is the current gradients and $G = -(g_1, g_2, \dots g_n)$ are a matrix containing the gradients associated to past tasks. This QP has a number of variable that is equal to the number of past tasks. The new gradients, that avoid catastrophic forgetting, is calculated as $\bar{g} = G^t v^* + g$. So basically the method force the new gradients to go in the same direction of the gradients associated to past tasks. This method requires more memory than a regularization approach such as EWC, and more time, but can work much better. The principal advantage is that while learning a new task also the old ones could be improved, improving the test accuracy on past data. GEM can be allocated between a rehearsal and a regularization approach.

A method that use rehearsal as well as dynamic architecture is called Evolvable Neural Turing Machine (ENTM) (proposed in [?]), it have agents that can store data by allocating additional memory. The structure of the network is found, starting from a base one, by dynamically change it during the optimization and the catastrophic forgetting are mitigated. An issue is that the agents allocate memory over time, without stopping, since no external constraints on the memory size are used.

Many more methods have been proposed, such as EXSTREAM ([5]) or ICALR ([19]), but wont be discussed here.

Chapter 5

Proposed approach: embedding regularization

In this section a novel rehearsal approaches will be introduced, the method is called called Embedding regularization. The idea is very simple and relies on the intrinsic capacity of a neural network to extract information from the input. Since the ANN elaborates the input data in order to extract and information, also called features, or embeddings, the idea is to force, given a past input, the ANN to keep the information extracted in the past as close as possible to the information extracted from the same past input but on a network trained on a new tasks. This means that if the two vectors are close, using some distance measure, then the ANN should be capable of classify correctly also past data; if the weights on the classification layer remains unchanged for the past tasks. The main advantage of this is that we don't need to manipulate gradients or do calculation based on past weights or similar, neither modify the network itself. In fact the correct working of this method is based on the fact that the ANN can manipulate the weights without constraints, since only the feature layer are used, but should keep the already stored information, and we don't care how the ANN does it.

The procedure is very simple: when a task i is over, a memory is filled with n triples $tm_i =< image_j, embedding_j, w_j >$, with $j = 1 \dots n$, where the images are draw from current task, the features vectors are computed using the network, and w is a weight associated to the triple and is used to sample from the memory. In each tasks, after the first one, the regularization step is performed after each batch, skipping the first. In the regularization step m triples are sampled from each past task according to the weights w. The weights can be initialized and modified using different methods that will be explained later. For each sampled triple $< image_k, embedding_k, w_k >$ the image $image_k$ are used to calculate a new features vector $embedding_{new}$, then the distance between $embedding_k$ and $embedding_{new}$ is calculated: $d_k = d(embedding_k, embedding_{new})$. This distance will be treated like a loss and will be minimized, so that the ANN needs to minimize the distance between the past and the current feature vector. The complete algorithm is 1

Algorithm 1 Embedding regularization

```
 1: procedure TRAIN PROCEDURE
 2:     Given a ANN
 3:     for each task $t_i$ in $T$ do
 4:         for each batch $b$ in $t_i$ do
 5:             Train the network on b
 6:             if $i > 1$ then EmbeddingDrive($i$)
 7:         SaveEmbeddings($t_i$)
 8: procedure EMBEDDINGDRIVE($i$)
 9:     for each task $j < i$ do
10:         extract m triples $< image_k, embedding_k, w_k >_j$
11:         Process the triples and extract the new embeddings $e_i$ for $i = 1 \ldots n$
12:         $m \leftarrow mean(\{d(embedding_i, e_i)\}_{for\ i=1 \ldots n})$
13:         back-propagate $\lambda m$
```

The algorithms contains many hyper-parameters. The first two are the memory size and the number of images m to sample in the embeddingDrive procedure. Also the parameter λ is very important, which controls the magnitude of the loss, in order to preventing it to dominate the one based on the current task; the best way to deal with the magnitude of the differences is to normalize the embeddings to a unit vectors, in this case we observed that the λ value is trivial to choose; in fact the default, and the best, choice is set λ to 1 and normalize the embeddings. Another parameter is the distance function: euclidean or cosine distance can be used; in the first case the constraints are more strong, the second one is less weak since it only force the new embedding to go in the same direction of old one, giving the network some , but some information can be lost in the process.

Another key part of the algorithm is the sample of the triples based on the weights. The most trivial approach is to have the same probability for each triple. Others methods are increment a counter each time the image are used, and set the weights as the inverse of it. A more sophisticated way is to assign an initial weight to each triple and, when a triple is processed, assign to the weight the current distance; this force the method to sample multiple times the images that are more difficult to remember. One more method consist in using an external network to extract structural information (features) from the the images, and weights the triples by the distance between the current task features and the ones extracted from the images in the memory; extracting structural information means that the network used to extract the features are never trained, in this way the weights change only when a new task is trained and the sample step will concentrate more on the triples that diverges more from the current task, since if the image is structurally difference from the another means that that is more probable for the network to forget the old information. The idea is inspired from [26].

The main idea of this method is to alleviate catastrophic forgetting by self adaption. The network itself should find a way to remember the past information not on a weight, or gradient, basis, but only knowing the features of the past images. in this way the ANN is more unconstrained and can be capable of keeping the past information while changing the weights during the current task training. Another key in the implementation is the speed: the algorithm is very fast and need very low number of image in memory to work; in facts the memory depends only on how many classes compose a task, and the sample size m are only needed to have an estimation of the distance from the past task to the current one, so a small number, around 10, can be sufficient.

Chapter 6

Experiments

6.1 MINST

The minst dataset is a collection of handwritten digits, introduced in [13], contains 70000 images, 10000 per digit. The images contains only white digits on black background, making the dataset a very easy benchmark for the current network methods; in fact, as the authors wrote: *It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on pre-processing and formatting.*

In this set of experiments a version called permuted minst ([11]), where each task contains the same images of the original dataset but the pixels are randomly permuted using a mapping (the same for each image). In this case we work in a data permutation scenario, where the output of the network remains of a fixed size during the training. Despite the fact that this is a easy benchmark, it is also a good starting point to compare different techniques to alleviate catastrophic forgetting.

In the first experiment an MLP composed of n ReLU layers of size M is used and the number of tasks are set to 4. In the embedding method the euclidean distance have been used, and the the images from different task are saved in the same memory, this lead do an higher number of triples to sample, 50 in this case, but since the problem is easy the results are good and the method still the quicker. The network was trained using SGD optimizer.

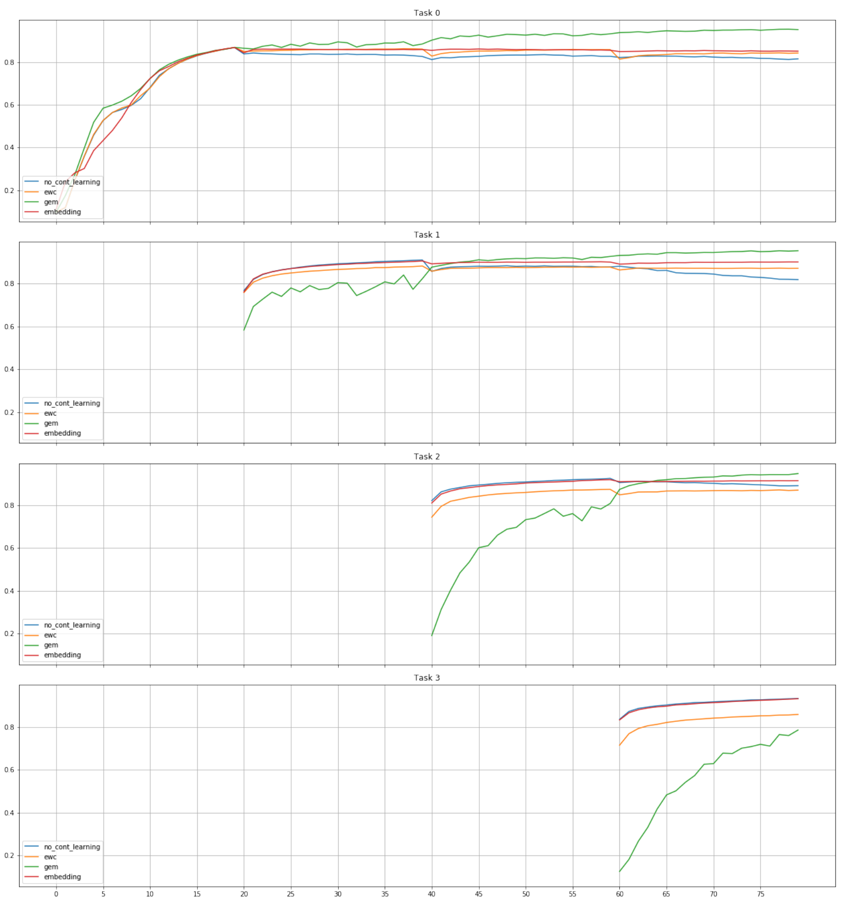

Figure 6.1. How the scores evolve during the training the ANN.

The image 6.1 shown the results of the training. We can see how all the methods performs well, even the training without any approach to alleviate catastrophic forgetting, because the problem is trivial. Can be notice that the GEM method improve the scores on the past task while learning the current one but the train on the current struggle to reach high scores while training. Our embedding method seems to lay, in terms of score, between the EWC and the GEM.

Since the embedding method works in the feature space of the images, is useful to see how it effects this space in order to empirically confirms our intuition behind the embedding method. In figure 6.2, which contains n rows (approaches) and m columns (tasks), we can see how the features space, for digits 0, 9 and 2, changes during the train phase; since the others approaches does not works in this space they are allowed to change and ignore it, our method keeps it basically the same.

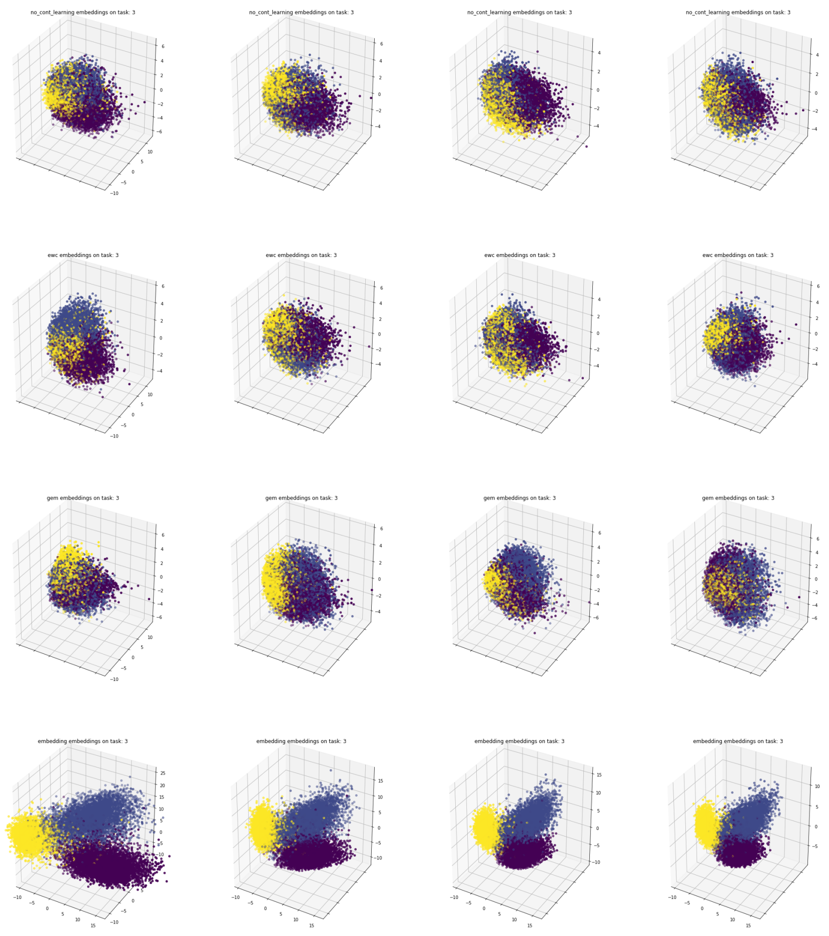

Figure 6.2. The feature space during the full training.

The second set of experiments involve the Kaf nets. In the first one the kaf networks are studied, in order to find the best kernel, while in the second one the multikaf are used and studied. The number of layers will be the same as the MLP version but the number of hidden neurons is only the 70% of the normal ANN approach. This reduces the number of parameter from 638810 to 396770.

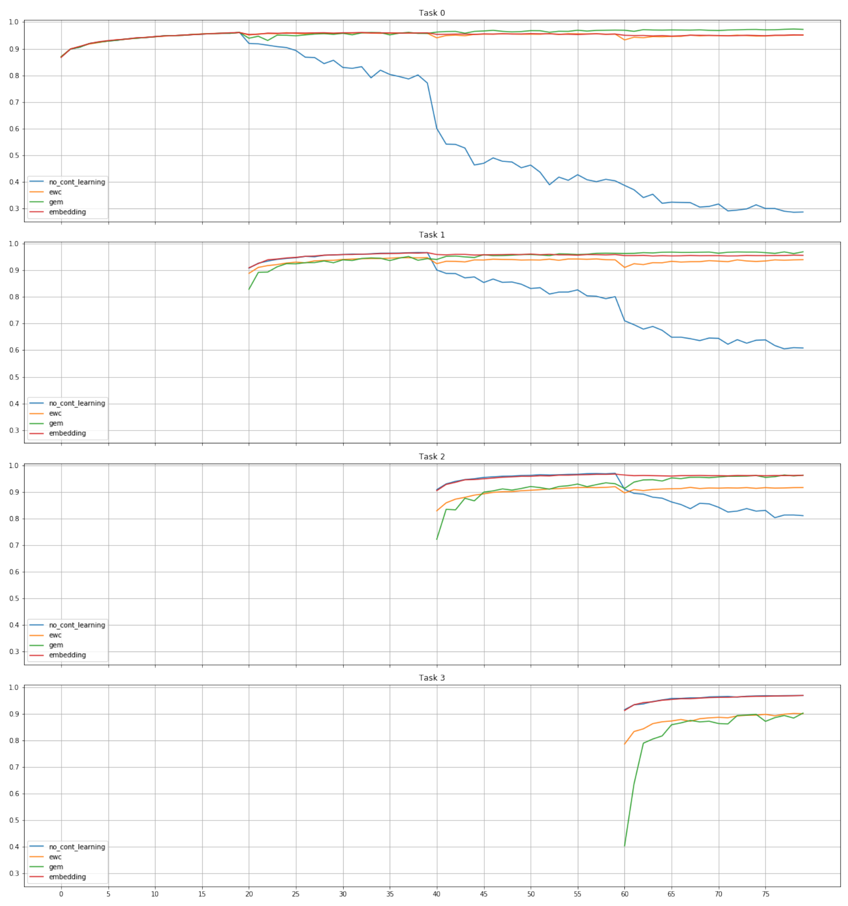

Figure 6.3. How the scores evolve during the training the Kaf net.

The first thing to be shown is a comparison between the kaf net and the ann. The image **??** contains the results obtained during the train phase, using the same asset of the train with the normal ANN. Two main differences can be noticed. The first one is that the train without any approach to alleviate catastrophic forgetting fails, this can be connected to the fact that the network change all the activation functions, leading to a completely forgetting of the past information. This can be connected to tendency of the Kaf nets to over-fit on the shape of the data. The second one is that train on a new task using the GEM approach leads to a faster adaption of the network, without forgetting the past information. In the end we can say that the kaf nets are better not only because need less parameters to perform well, but also because can outperform the normal ANN approach, using the right approach.

6.2 CIFAR10

6.3 CIFAR100

Bibliography

[1] Fabio Aiolli and Michele Donini. Easymkl: A scalable multiple kernel learning algorithm. *Neurocomputing*, 169, 04 2015.

[2] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2016.

[3] George Cybenko. Approximation by superpositions of a sigmoidal function. *MCSS*, 2:303–314, 1989.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[5] Tyler L. Hayes, Nathan D. Cahill, and Christopher Kanan. Memory efficient experience replay for streaming learning. *CoRR*, abs/1809.05922, 2018.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[7] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

[8] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[9] Kurt Hornik. Hornik, k.: Approximation capabilities of multilayer feedforward network. neural networks, 251-257. *Neural Networks*, 4, 01 1991.

[10] Ronald Kemker, Angelina Abitino, Marc McClure, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. *CoRR*, abs/1708.02072, 2017.

[11] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *CoRR*, abs/1612.00796, 2016.

[12] Y. Le Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard. Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, Nov 1989.

[13] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits, 1998.

[14] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40:2935–2947, 2018.

[15] Xialei Liu, Marc Masana, Luis Herranz, Joost van de Weijer, Antonio M. López, and Andrew D. Bagdanov. Rotate your networks: Better weight consolidation and less catastrophic forgetting. *CoRR*, abs/1802.02950, 2018.

[16] Vincenzo Lomonaco and Davide Maltoni. Core50: a new dataset and benchmark for continuous object recognition. *CoRR*, abs/1705.03550, 2017.

[17] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continuum learning. *CoRR*, abs/1706.08840, 2017.

[18] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

[19] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, and Christoph H. Lampert. icarl: Incremental classifier and representation learning. *CoRR*, abs/1611.07725, 2016.

[20] A. Robins. Catastrophic forgetting in neural networks: the role of rehearsal mechanisms. In *Proceedings 1993 The First New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*, pages 65–68, Nov 1993.

[21] ANTHONY ROBINS. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[22] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *CoRR*, abs/1606.04671, 2016.

[23] Syed Shakib Sarwar, Aayush Ankit, and Kaushik Roy. Incremental learning in deep convolutional neural networks using partial network sharing. *CoRR*, abs/1712.02719, 2017.

[24] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 1257–1264. MIT Press, 2006.

[25] Sebastian Thrun and Tom M. Mitchell. Lifelong robot learning. *Robotics and Autonomous Systems*, 15(1):25 – 46, 1995. The Biology and Technology of Intelligent Autonomous Agents.

[26] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Deep image prior. *CoRR*, abs/1711.10925, 2017.

[27] Friedemann Zenke, Ben Poole, and Surya Ganguli. Improved multitask learning through synaptic intelligence. *CoRR*, abs/1703.04200, 2017.