GrNet : a neural network architecture for manifold learning

Author
Mohammed HSSEIN

Supervisors
Rémy BOYER
Jérémie BOULANGER

Academic year 2019/2020
promo : 2021

Acknowledgements

I would like to thank my supervisor Mr. Rémy BOYER, first for proposing this beautiful subject, the subject was new for me, I learned a lot from it and it allowed me to apply a lot of the things that I have learned.

I want to thank him especially for the consistent supervising during this period of two months. He kept pushing me towards things I was not sure I am capable of. Thank you Sir for giving from your precious time to meet me every week.

This work would not have been done without, first your previous brilliant papers, and second, without the ideas that you have helped me with.

I hope this work will be a subject of satisfaction for you even though we did not reach some of the goals we set.

I want to thank also Mr. Jérémie BOULANGER, for his assistance each week and his remarks about papers. These remarks helped me to be comfortable while reading the main papers from the literature and understanding some of the difficult proofs and results and it certainly helped me to establish the main theoretical result of this work. I want to thank also phd student Ouafae KARMOUDA, for sharing some insightful lectures and papers of her work.

Summary

1 Introduction

Many relevant problems in modern machine learning nowadays involve subspace-structured features, orthogonality or low-rank constrained objective functions, or subspace distances. These characteristics are expressed matematically with the use of Grassmann manifolds.

Recent papers about the topic of **Deep learning for computer vision** [1], [4] for instance, and [2] on which this work is based, have integrated **new blocs** or replaced classic ones in classical deep neural network architectures, for instance (*CNNs*). The main objective is to adapt the classical architectures to other type of data inputs : for example ***Grassmannian data***. The learning procedures then, (**optimisation, update of weights, ...**) are performed on specific manifolds (*usually projected stochastic gradient descent*), for example manifold of **fixed rank matrices**, **Stiefel manifold**, ... and so one and so forth.

The term **Grasmannian data** might be frightening at first. it is just an operation on the initial standard data, called also **euclidean data**. This operation, in the sake of simplicity, is here, the application of **Principal Component Analysis** to the original data-set to obtain a more compact representation of it. as result the data relies on a specific mathematical manifold. PCA is trying the find the directions such that the projection of the data on these directions has the highest variance [6].

In this simple work, we will review block-by-block the architecture proposed in [2], and then implement it step by step before delving into a discussion of results and problems encountered in the training procedure. The data-set used in the tests is the classic **MNIST** data-set of handwritten digits. Although, the data is first transformed into a Grassmannian data-set so the most important features are kept. Of course a tuning operation is needed to decide of the dimensions of the preprocessed images before feeding them to the network.

2 Purpose of the internship

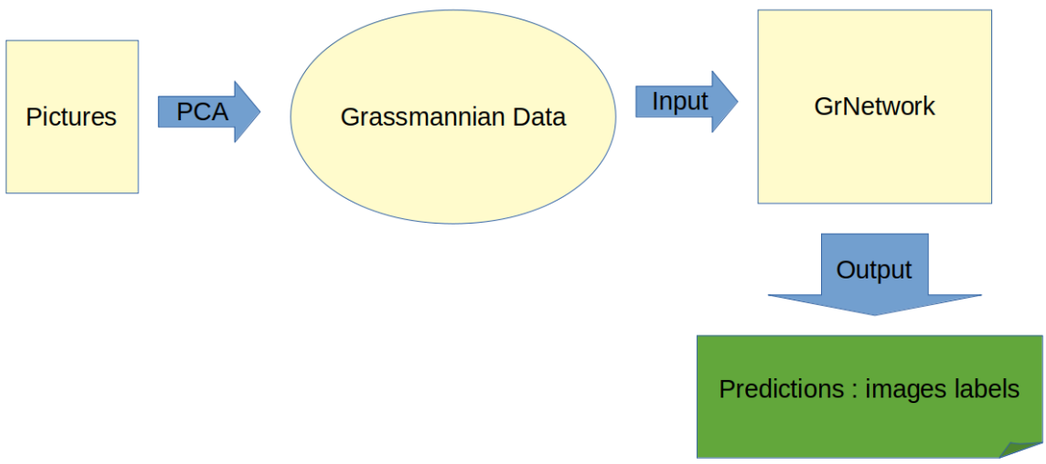


Figure 1: The purpose of the internship

The scheme in figure 1, explains the purpose of this internship. We start from an euclidean data, the MNIST data-set, we apply a PCA decomposition to obtain a Grassmannian representation of the data-set. This data is feed to the network, which will output the predictions, ie the labels of the input pictures. We will evaluate this architecture and how it can handle the Grassmannian data. A study can be conducted after this preliminary, to make a generalization of the architecture into tensors of order 3. After this step, using the **Canonical polyadic decomposition**, we cans generalize to input data of higher dimensions.

3 Grassmann manifolds : overview

3.1 Definition

The Grassmann manifold $Gr(n, k)$ with the condition $n \geq k > 0$, is the set of k-dimensional linear sub-spaces embedded in an n-dimensional linear space (either real or complex). Mathematically speaking, this set is defined as :

$$Gr(n,k) = \left\{ \text{span}(\mathbf{X}) : \mathbf{X} \in \mathbb{R}^{n \times k}, \mathbf{X}^T\mathbf{X} = \mathbb{I}_k \right\} \tag{1}$$

where, $\mathbb{I}_k$ is the identity matrix of size k, and the name *span*, is a linear space, of all the linear combinations of row-vectors of X, mathematically speaking :

$$\text{span}(\mathbf{X}) = \left\{ \sum_{i=1}^{k} \lambda_i V_i : k \in \mathbb{N}, \lambda_i \in \mathbb{R}, V_i \ i^{th}\text{column of } \mathbf{X} \right\} \tag{2}$$

A basic representation of $Gr(3, 1)$ is as shown in the figure 1 :

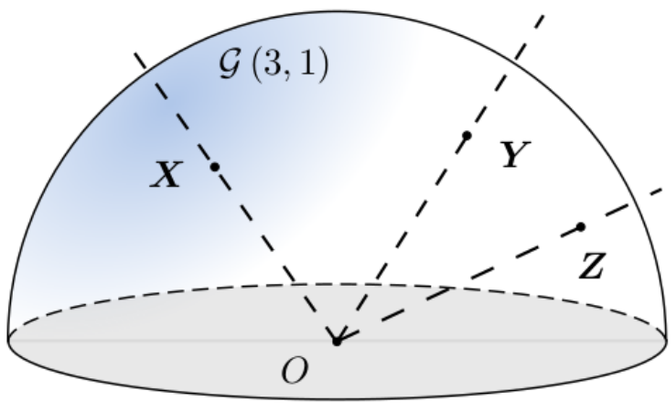


Figure 2: The set $Gr(3, 1)$ of the space $\mathbb{R}^{3\times3}$.$\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ are points on the manifold.

For more details about the distances and geodesics in the manifold, please refer to [1], as the work here is not to delve into the mathematics in detail behind the topological structure of Grasmannian manifolds.

3.2 How to obtain Grassmannian data

Well, obviously, most of standard data the we encounter : images, audio-records, ... are not Grasmannian data. The aim of Grasmann manifolds is to obtain a compact representation of the standard (euclidean) data. Let's take an example of handwritten digits MNIST classic data-set : an image is a sparse image of huge number of zeros, and the number of pixels containing useful information about the shape of digit constitutes nearly 30% of the picture. Let's take another example : an RGB image of a beautiful lake of size (1080, 1920, 3). After applying the PCA, it becomes an (1080, 370, 3) image, and yet it still contains more than 96% of the information contained in the original image.

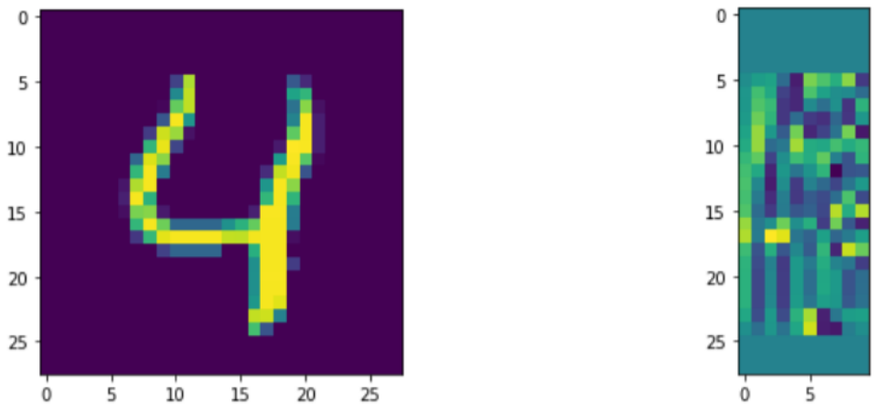


Figure 3: Original image and the orthonormal component (Grassmannian)

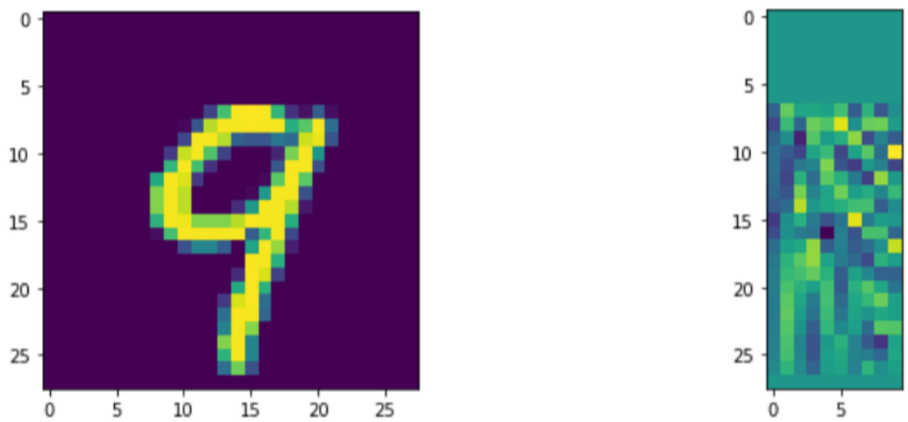


Figure 4: Original image and the orthonormal component (Grassmannian)

we can see the images obtained after the PCA, are not meaningful for humans, although, for machines, those images still have an accurate representation of original ones (here PCA done so that the obtained images contain 96% of variance of the original data.

4 GrNetwork : a novel neural network architecture

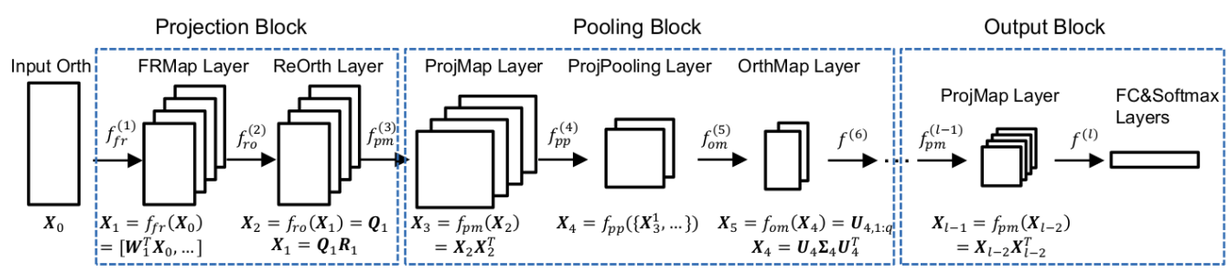


Figure 5: GrNetwork architecture

The GrNet architecture consists of tree blocks : **Projection Block**, **Pooling block**, and finally **Output block**.

The **Projection Block** is composed of two layers : **FRMap Layer** and **ReOrth Layer**. The input (Grassmannian) matrix $\mathbf{X}_0$ is feed to the network. The **FRMap** layer, performs a filtering operation : it applies some filters $F = \{\mathbf{W}_1, \mathbf{W}_2, ..., \mathbf{W}_m\}$ to the input matrix. The filters are on the manifold of row full rank matrices (fixed rank matrices), defined as :

$$\mathrm{M} = \left\{ \mathbf{X} \in \mathbb{R}^{n \times k} \ / \ \mathrm{rank}(\mathbf{X}) = r \right\} \tag{3}$$

The set F is equivalent, in classic deep learning architectures, to convolution part where some filters have to be applied to the image in order to extract relevant image features. This is an example in which the learning procedure of the filters will be done on the manifold M.

As the previous operation holds matrices that aren't relying on the Grassmann manifold, the **ReOrth** layer, does the job here: it performs a **QR** decomposition and then keeps just the orthogonal resulting matrix $\mathbf{Q}$ that verifies $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. Intuitively, this means we keep the vectors of a new space in which the data has been projected. This ends the first block. To make the link between this architecture, and classic ConvNets, here the **QR** decomposition has the role of **ReLu**-like layers, ie applying a non linear activation to the input data.

Note that **QR** decomposition is used in computer vision in the field of background substraction and automatic removal of objects from images or videos : we divide the videos into frames and perform **QR** decomposition on each one, and then keep $\mathbf{Q}$ term. After reconstruction of frames, the video appears without the desired object. A simple example is shown here of the numbers 5 and 0 from the **MNIST** data-set, just as an illustration, as it is not the subject here.

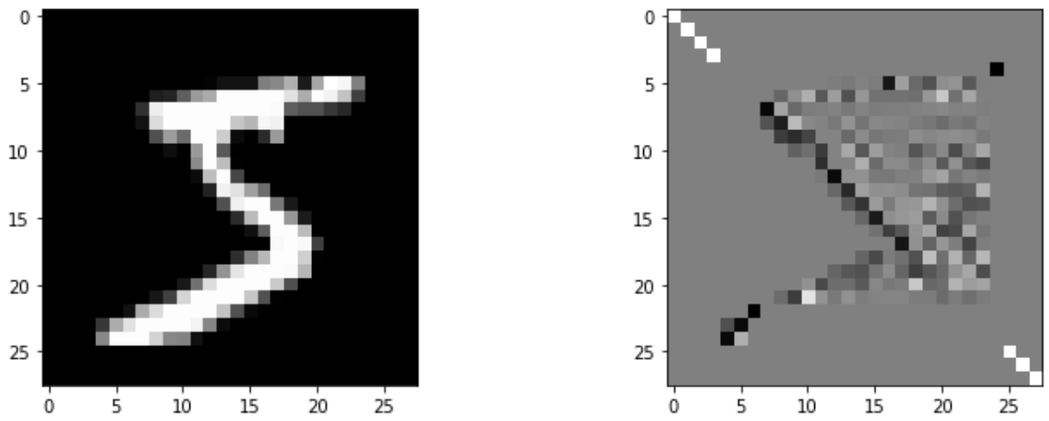


Figure 6: original picture and $\mathbf{Q}$ component after **QR** decomposition

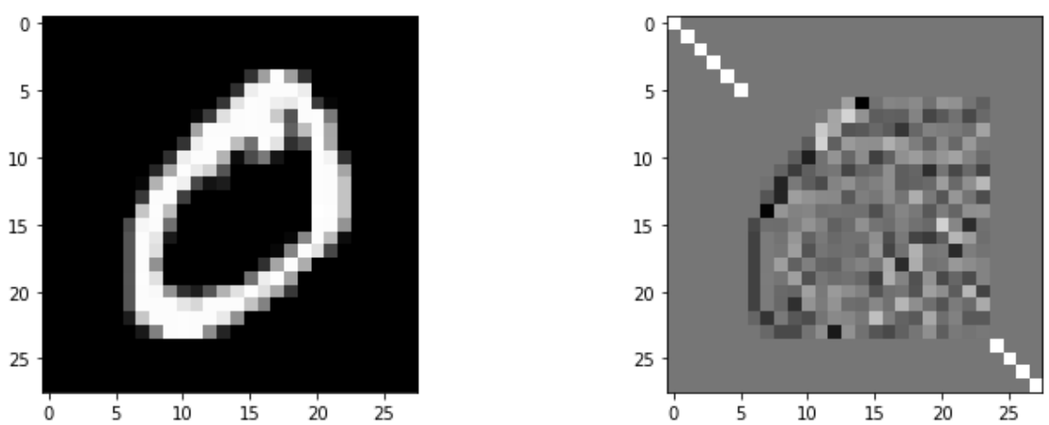


Figure 7: original picture and $\mathbf{Q}$ component after **QR** decomposition

The **Pooling Block**, is the equivalent of the he **Pooling** layer in CNNs. the goal is to reduce the complexity of the inputs, by producing a more compact data.

ProjMap layer, we transform the matrices obtained in the previous layer to some matrices lying in the SPD manifold, thus the operations : $\mathbf{X}^T\mathbf{X}$. There are m matrices (remember m filters used in **FrMap** Layer. Please refer to [2] for more insights about this transformation.

Many types of pooling are possible for the **ProjPooling** layer : for more information refer to [2]. In this work, we will implement the **mean-pooling**, ie taking the mean of all the matrices coming from the **ProjMap** layer.

Matrices obtained after **ProjPooling** layer, aren't on the Grassmannian manifold, that why performing a reorthonormalization process is needed. It's the role of **OrthMap** layer. we perform an **singular value decomposition**. we then tronque it and keep just the first q columns of U corresponding to the first q singular values in an non-decreasing order. Note that q here is the second dimensional of $Gr(n, k)$, meaning k in this notation.

We project then the result on the manifold of SPD matrices. This is the last block. The **ProjMap** layer does the job as we have seen. In this stage, one can repeat the previous three blocks as many times as we want, before feeding the result to the final **Fully connected** layers.

5 Implementation of GrNetwork

For the implementation, let's recall the equations for forward pass and the back-propagation. Some nice equations are here to be implemented. But not that much when it comes to the gradients: the gradients are much more complicated and needs some advanced calculus to be performed on matrices, and for this reason we will use a deep learning framework : Pytorch. For those looking for proof of these equations, you can refer to the outstanding article [4].

5.1 Forward pass

The equations for the **Forward pass** are :

- **FrMap layer** :

$$\mathbf{X}^1 = \left\{\mathbf{X}_1^1, \mathbf{X}_2^1, ..., \mathbf{X}_m^1\right\} = f_{fr}(\mathbf{X}_0) = \left\{\mathbf{W}_1\mathbf{X}_0, \mathbf{W}_2\mathbf{X}_0, ..., \mathbf{W}_m\mathbf{X}_0\right\} \tag{4}$$

  which is equivalent to write for each $i = 1, 2, ..., m$: $\mathbf{X}_1^i = \mathbf{W}_i\mathbf{X}_0$

- **ReOrth layer** :

$$\mathbf{X}^2 = \left\{\mathbf{X}_1^2, \mathbf{X}_2^2, ..., \mathbf{X}_m^2\right\} = f_{ro}(\mathbf{X}_1) = \left\{\mathbf{Q}_1, ..., \mathbf{Q}_m\right\} \tag{5}$$

  where $\mathbf{X}_i^2 = \mathbf{Q}_i\mathbf{R}_i$ is the QR decomposition of matrix $\mathbf{X}_i^2$ for $i = 1, 2, ..., m$. Note that here matrix aren't on exponent 2, but is just a notation. The matrices aren't even square so the exponent as known cannot be defined ...

- **ProjMap layer** :

$$\mathbf{X}^3 = \left\{\mathbf{X}_1^3, \mathbf{X}_2^3, ..., \mathbf{X}_m^3\right\} = f_{pm}(\mathbf{X}^2) = \left\{\mathbf{X}_1^{2^T}\mathbf{X}_1^2, \mathbf{X}_2^{2^T}\mathbf{X}_2^2, ..., \mathbf{X}_m^{2^T}\mathbf{X}_m^2\right\} \tag{6}$$

  which means we perform the product of each matrix $\mathbf{X}_i^2$ by it's transpose matrix $\mathbf{X}_i^2$, for $i = 1, 2, ..., m$.

- **ProjPooling layer** :

$$\mathbf{X}^4 = f_{pp}(\mathbf{X}^3) = \frac{1}{m}\sum_{i=1}^{i=m}\mathbf{X}_i^3 \tag{7}$$

  we perform an arithmetic mean of the matrices $\mathbf{X}_i^3$ for $i = 1, 2, ..., m$. Note that to reduce complexity we can choose just n matrices among m.

- **OrthMap layer** :

$$\mathbf{X}^5 = f_{om}(\mathbf{X}^4) = \mathbf{U}_{1:q}^4, \quad \mathbf{X}^4 = \mathbf{U}^4\Sigma\mathbf{U}^{4^T} \tag{8}$$

  Here we keep the first q eigen vectors obtained by the SVD decomposition, ordered in non-decreasing way. Recall that q is the second shape of Grassmannian input matrix $\mathbf{X}_0$, of shape (d_0, q). Matrix obtained $\mathbf{X}^5$ is then an orthogonal matrix.

- **ProjMap layer** :

$$\mathbf{X}^6 = f_{pm}(\mathbf{X}^5) = \mathbf{X}^{5^T}\mathbf{X}^5 \tag{9}$$

  in the previous step, the matrix was orthogonal, but here we apply the matrix product by the transpose so the output remains in the manifold of SPD matrices.

- **Fully connected layer** :

$$\mathbf{FC} = \mathbf{b} + \mathbf{WA} \tag{10}$$

where $\mathbf{b}$: bias, $\mathbf{W}$: weights, $A = \mathbf{X}^6$ flattened. The term flattened stands here for an operation of vectorizing the matrix $\mathbf{X}^6$ of shape (d_1, d_1) to transform it to a vector of shape $(d_1^2, 1)$. Bias b and weight matrix $\mathbf{W}$ are of shape respectively : $(C, 1)$ and (C, d_1^2) with C number of classes, here $C = 10$.

- **Final output and predictions**

  For the final outputs, we have the FC vector, we apply the softmax function to it. softmax function is defined as :

$$\text{Softmax}(\mathbf{X}) = \left(\frac{e^{x_1}}{\sum_{i=1}^{C} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{C} e^{x_i}}, ..., \frac{e^{x_C}}{\sum_{i=1}^{C} e^{x_i}} \right) = \left(\frac{e^{x_j}}{\sum_{i=1}^{C} e^{x_i}} \right)_{1 \leq j \leq C} \tag{11}$$

  Where $\mathbf{X} = \Big[x_1, x_2, ..., x_C \Big]^T$.

For the predictions then, we have to look for the maximum value in the output, and then the index of this value in the output matrix is the predicted class. Mathematically, if $x_i = \mathbf{max}(\mathbf{X})$, **then i is the predicted class**. Intuitively, the **Softmax** gives the **probability of each class. The class likly to be the true one, is the class with highest probability**.

5.2 Backward pass

5.2.1 Equations

The equations for the **back-propagation** are :

- **FrMap layer** $(i = 1)$

$$\frac{\partial \mathcal{L}^{(i)}}{\partial X^{i-1}} = \frac{\partial \mathcal{L}^{(i+1)}}{\partial X^i} X_{i-1}^T \tag{12}$$

  where $\frac{\partial \mathcal{L}^{(i+1)}}{\partial X^i}$ is the gradient of newt layer (OrthMap) and $\mathbf{X}_{i-1}$ input matrix for FrMap.

- **ReOrthMap** $(i = 2)$

$$\frac{\partial \mathcal{L}^{(i)}}{\partial X^{i-1}} = \left(S^T \frac{\partial \mathcal{L}^{(i+1)}}{\partial X^i} + Q \left(Q^T \frac{\partial \mathcal{L}^{(i+1)}}{\partial X^i} \right)_{bsym} \right) \left(R^{-1} \right)^T \tag{13}$$

  Here $\frac{\partial \mathcal{L}^{i+1}}{\partial X^i}$ denotes the gradients from the next layer, ie ProjMap layer. Here we can see the interest in PyTorch. We will discuss its role in next section of this report. Note also that in this context : $\mathbf{A}_{bsym} = \mathbf{A}_{tril} - (\mathbf{A}^T)_{tril}$ and $\mathbf{A}_{tril}$ extracts the elements below the diagonal of $\mathbf{A}$.

- **ProjMap** $(i = 3)$

$$\frac{\partial \mathcal{L}^{i}}{\partial X^{i-1}} = \frac{\partial \mathcal{L}^{i+1}}{\partial X^i} \frac{\partial X^i}{\partial X^{i-1}} \tag{14}$$

  and $\frac{\partial X^i}{\partial X^{i-1}}$ is calculated by pytorch automatically. Note also $\frac{\partial \mathcal{L}^{i+1}}{\partial X^i}$ stands for the gradient from the ProjPooling layer since layer i stands for actual layer. Now we reiterate again :

- **ProjPooling** $(i = 4)$

$$\frac{\partial \mathcal{L}^{i}}{\partial X^{i-1}} = \frac{\partial \mathcal{L}^{i+1}}{\partial X^i} \frac{\partial X^i}{\partial X^{i-1}} \tag{15}$$

  and $\frac{\partial X^i}{\partial X^{i-1}}$ is calculated by pytorch automatically. Note also $\frac{\partial \mathcal{L}^{i+1}}{\partial X^i}$ stands for the gradient from the next layer ie OrthMap layer and i stands for actual layer.

- **OrthMap Layer** $(i = 5)$

$$\frac{\partial \mathcal{L}^{(i)}}{\partial X} = U \left(K^T \circ \left(U^T \begin{bmatrix} \frac{\partial \mathcal{L}^{(i+1)}}{\partial X} & O \end{bmatrix} \right) \right) U^T \tag{16}$$

Here $\frac{\partial L^{i+1}}{\partial X}$ denotes the gradients coming from the next layer, ie Fully connected layer. the operator $\circ$ denotes here famous ***hadamard*** product between two matrices. $\left[\frac{\partial \mathcal{L}^{(i+1)}}{\partial X} \quad O\right]$ is matrix formed by concatenation of two matrices of shapes (d_1, q) and $(d_1, d_1 - q)$ respectively. Last matrix is a zero matrix.

- **Fully connected Layer** $(i = 6)$

  In this part, we should compute the gradient of the loss wrt the samples. This will provide the quantity $\frac{\partial \mathcal{L}^{i+1}}{\partial X}$ present in previous formula of **OrthMap Layer**, and then the equations can be completed and thus we can perform the learning step by minimizing the loss, with a projected stochastic gradient descent. This is finally the meaning of manifold-based optimization, ie, the learning process, (optimisation of weights $\{\mathbf{W}_1, \mathbf{W}_2, ..., \mathbf{W}_m\}$ is performed in a specific manifold).

  The fully connected layer can be divided in two parts : first, we vectorize the matrix $\mathbf{X}^6$, and second we perform the product of equation (10).

  - **Fully connected Layer** $(i = 7)$

$$\mathbf{X}^7 = vect(\mathbf{X}^6) \tag{17}$$

  where X^6 is matrix defined in equation 9. this operation of vectorizing is described in comments below equation 10.

  - **Fully connected Layer** $(i = 8)$

$$\mathbf{X}^8 = \mathbf{X}^7\mathbf{W} + \mathbf{b} \tag{18}$$

  The gradients for these two parts are : $(\ i = 7, i\ = 8)$

$$\frac{\partial \mathcal{L}^i}{\partial X^{i-1}} = \frac{\partial \mathcal{L}^{i+1}}{\partial X^i}\frac{\partial X^i}{\partial X^{i-1}} \tag{19}$$

  To end the equations of back-propagation, it remains to evaluate the gradient in equation 18 for $i\ = 8$. One gradient remains to be calculated is $\frac{\partial \mathcal{L}^9}{\partial X^8}$. Recall the fact that X^7, X^8, W, and b are vectors. $\mathcal{L}^9$ here is the (final stage) loss l that will we defined later written in matrix form. Let's define first the loss, and then establish the analytical equations for this purpose.

The classic loss for classification problems in deep learning is **categorical cross-entropy loss** defined as :

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{C}\sum_{j=1}^{C}\left[\hat{y}_j \log(y_j) + (1 - \hat{y}_j)\log(1 - y_j)\right];\ y = (y_j)_{1\leq j\leq C},\ \hat{y} = (\hat{y}_j)_{1\leq j\leq C} \tag{20}$$

where y is the softmax output from the network, and $\hat{y}$ is the target vector, ie the vector from the class labels. Note that if $\hat{y}$ represents the k^{th} class among the C classes, then it is a zero vector except for the k^{th} element where it equals 1.

Now to perform the training, we should compute the last gradient. from which we will first update $\mathbf{W}$ and the bias $\mathbf{b}$, and then back-propagate to the network until reaching the filters. This part of updating $\mathbf{W}$ and $\mathbf{b}$ could be done by pytorch in one line. Although we will give these equations in detail.

Right now we should compute $\frac{\partial l}{\partial h}$ for $h = \mathbf{b},\ \mathbf{W},\ \mathbf{X}^8$. Note that :

$$\frac{\partial \mathcal{L}^9}{\partial h} = \frac{\partial l}{\partial h} = \left(\frac{\partial l}{\partial h_a}\right)_{1 \leq a \leq C} \tag{21}$$

Now starting from the definition, a simple calculus leads to the equation :

$$\frac{\partial l}{\partial h_a} = -\sum_{j=1}^{j=C} \left(\frac{\hat{y}_j}{y_j} - \frac{1-\hat{y}_j}{1-y_j}\right) \frac{\partial y_j}{\partial h_a} \tag{22}$$

So we have just to compute gradient in equation 22.
For $h = \mathbf{X}^8$, we have :

$$\frac{\partial y_j}{\partial h_a} = \frac{\partial y_j}{\partial x_a} = \frac{\partial \text{Softmax}(x_j)}{\partial x_j} \frac{\partial x_j}{\partial h_a} = \text{Softmax}^{'}(x_j)\delta_{a,j} = S^{'}(x_j)\delta_{a,j} \tag{23}$$

where $\delta_{a,j}$ is the Kronecker's symbol. If we denote Softmax by S, then we have the a^{th} element of $\frac{\partial \mathcal{L}^9}{\partial \mathbf{X}^8}$. So far this task is finished. Note that with simple calculation : $S^{'}(x_j) = S(x_j)(1 - S(x_j))$, and it yields :

$$\frac{\partial l}{\partial x_a} = -\sum_{j=1}^{j=C} \left(\frac{\hat{y}_j}{y_j} - \frac{1-\hat{y}_j}{1-y_j}\right) S^{'}(x_j)\delta_{a,j} \tag{24}$$

Now that we have finished the equations of the back-propagation for the filters, and gave formula of $\frac{\partial \mathcal{L}^9}{\partial \mathbf{X}^8}$, let's move to compute $\frac{\partial \mathcal{L}^9}{\partial \mathbf{W}}$. and $\frac{\partial \mathcal{L}^9}{\partial \mathbf{b}}$, and then give equations of updating W and b. But this is quit straightforward according to equation 22. We have as a result :

$$\frac{\partial y_j}{\partial w_{a,b}} = S^{'}(x_j)\frac{\partial x_j}{\partial w_{a,b}} = S^{'}(x_j)x_{1a}w_{a,b}\delta_{b,j} \tag{25}$$

where $\mathbf{X}^7 = \left(x_{1,a}\right)_{1 \leq a \leq d_1^2}$ is the $\mathbf{X}^7$ vector in the forward pass. Finally we have :

$$\frac{\partial l}{\partial w_{a,b}} = -\sum_{j=1}^{j=C} \left(\frac{\hat{y}_j}{y_j} - \frac{1-\hat{y}_j}{1-y_j}\right) S^{'}(x_j)x_{1a}w_{a,b}\delta_{b,j} \tag{26}$$

and with the same procedure, we have the gradient w.r.t the bias :

$$\frac{\partial l}{\partial b_a} = -\sum_{j=1}^{j=C} \left(\frac{\hat{y}_j}{y_j} - \frac{1-\hat{y}_j}{1-y_j}\right) S^{'}(x_j)\delta_{a,j} \tag{27}$$

5.2.2 updating bias and weight for output layers

We will adopt the Stochastic gradient descent algorithm :

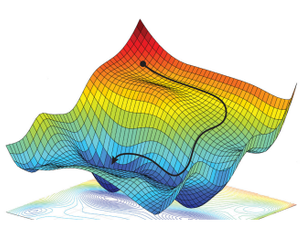


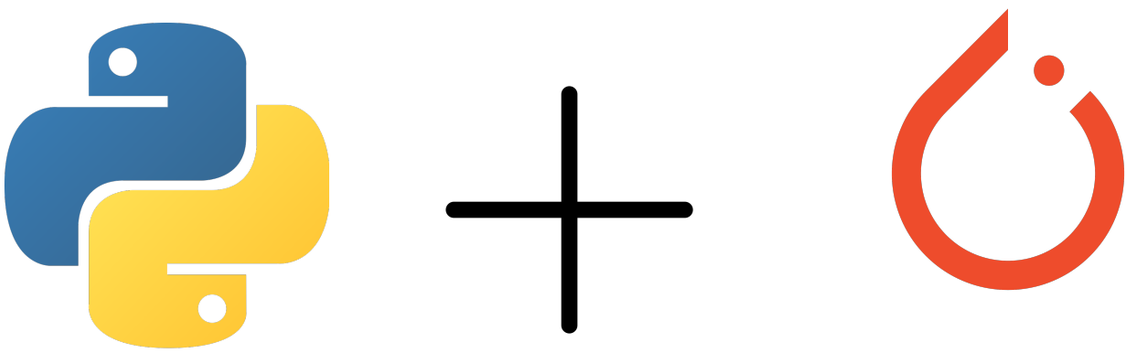
Figure 8: SGD : looking for the minimum

The learning procedure will be (see discussion section for more specifications):

$$\mathbf{H} = \mathbf{H} - \alpha \mathbb{E}_{\mathbf{y}} \nabla_{\mathbf{H}} \mathcal{L} \text{ where } \; H = \mathbf{W}, \; H = \mathbf{b} \tag{28}$$

5.3 Implementation : Python + Pytorch

For the implementation, we'll use python as a programming language, and we'll build the architecture using the famous deep learning framework pytorch.



5.3.1 Python Ok .. But what is Pytorch and why ?

Pytorch is a deep learning framework, developed by Facebook's AI research lab. It comes with a class of tensors called **torch tensors** to store and operate on homogeneous multidimensional rectangular arrays of numbers. Torch tensors are similar of those of **NumPy** library, but with the possibility to run on GPU(s). Another incentive to use Pytorch, is the **Auto-grad** module. A recorder records operations performed on pytorch tensors during the forward pass, and then is capable of computing the gradients, for the backward pass, which is something very useful when it comes to deep learning. So, when it comes to matrix multiplication operations, we don't have to worry too much about the gradients. This figure shows the forward and the backward pass

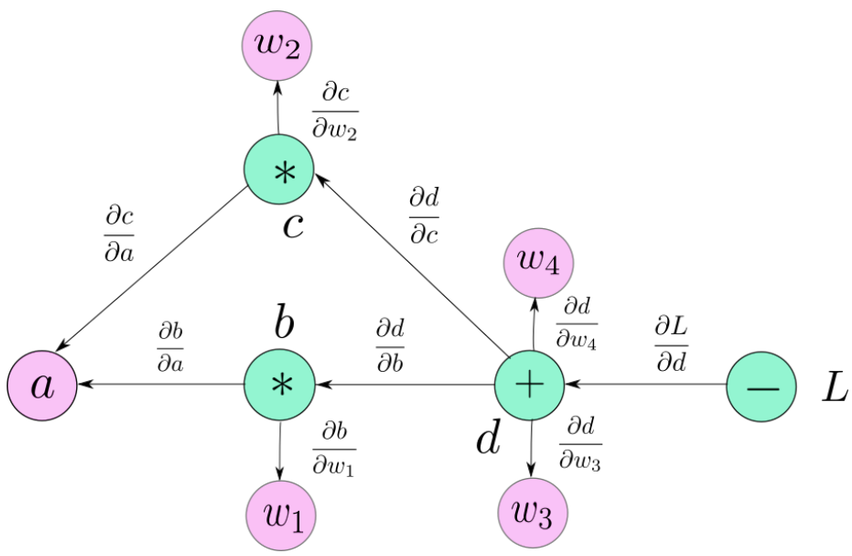


Figure 9: **Pytorch graph** : forward and backward pass example

operations. For the forward pass we compute the result L, and for the backward pass pytorch computes the gradients w.r.t of any stage (layer).

5.3.2 The data-set

The data-set used is the famous MNIST data-set of hand written digits. it contains 10 classes. It is split into training set and test set. The training set is composed of 60000 image of $(28, 28)$ pixels, and the test est of 10000 images of each class. Note that the training data-set is quit balanced, but not perfectly balanced. But the data here is note Grassmannian data. First we divide each pixel of each image by 255 to get values of matrices between 0 and 1.

Second, we perform a PCA decomposition, and we keep only first 10 eigenvectors of the 10 first highest eigenvalues. with value of 10, we conserve 95% of the data variance, as shown in the figure 8:

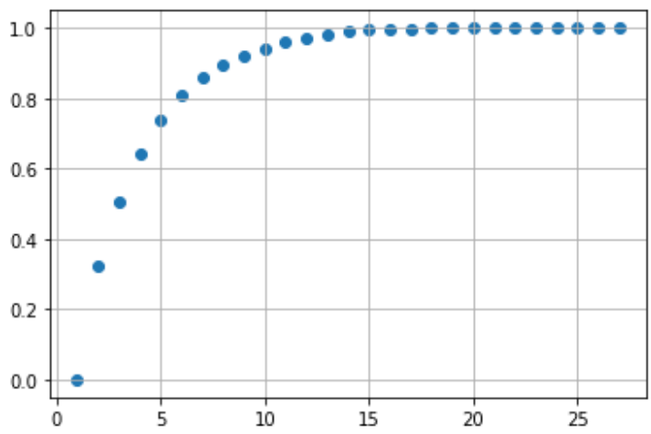


Figure 10: **PCA** decomposition : variance and num of columns ot keep

5.3.3 Code

We will use the classic oriented object programming paradigm. First, we create a class **GrNetwork**, from which we will instantiate an object **model** for the training and testing. We will create a separated file named **utils.py** where we code some functions needed for the architecture. For example **Pooling mean** function, **call_reim_grad** function to transform the euclidean gradient to Reimannnian gradient before updating the filters, and two classes **OrthMapLayer** and **ReOrthMapLayer** respectively, to perform the **QR** and **SVD** decomposition, and to compute the gradients in equations 12 and 13. code implementation could be found in my github repository at : `https://github.com/Mohammed-Hssein/GrNet/`. I will mention just a one part a bit confusing about the transformation of euclidean gradients to Reimannian gradients. This transformation is shown in code below : for line 12 of this code, we output the Reimannian vector. For updating weights, we need a retraction to project the weights onto the manifold of fixed rank matrices. But if we perform the projection without dividing with norm, the weights grow exponentially with epochs and the training ends in epoch 10 because the SVD algorithm cannot converge anymore. This procedure of dividing by the norm, ensures the values of weights keeps values between 0 and 1.

```
1  import numpy as np
2  def call_Reimann_grad(W, EucGrad):
3      """
4      W : weight to be updated
5      EucGrad : euclidean gradient
6      """
7      EucGradT = EucGrad.astype(np.double).transpose()
8      W = W.astype(np.double)
9      U, _, V = np.linalg.svd(np.dot(W, EucGradT))
10     Q = np.dot(V, U.transpose())
11     Rgrad = np.dot(EucGradT, Q) - W.transpose()
12     return Rgrad.astype(np.double)
```

Listing 1: Reimannian grad from euclidean grad(*sorry for mixing camel Case and under_score notations for function names !!!*)

This is how the authors of [2] used, based on the ***manopt*** MATLAB library in `https://www.manopt.org/reference/manopt/manifolds/symfixedrank/symfixedrankYYfactory.html`. For the details, you can look lines of `https://github.com/zhiwu-huang/GrNet/blob/master/grnet_train_afew.m` , lines from 183 to 189.

6 Analysis

6.1 Training

We choose these parameters for the training : learning rate $\epsilon = 10^{-2}$, batch-size = 30, and as number of epochs 100. This means we will pass the training data-set throw the model 100 times during the training period. The batch-size, means we will pass the data by batches, not image by image. Thus, for the loss, we will not get single loss for each image and sum them up, but a mean loss for each batch size, and sum up the losses for one batch (The update is performed once each batch-size during the training). For more details, you can see documentation of pytorch here `https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html`.

6.2 Results

We have performed two studies : accuracy in function of number of filters, and accuracy as function of number of ANN layers in the output. in each case we draw the evolution of loss during the training.

Results are shown in these figures :

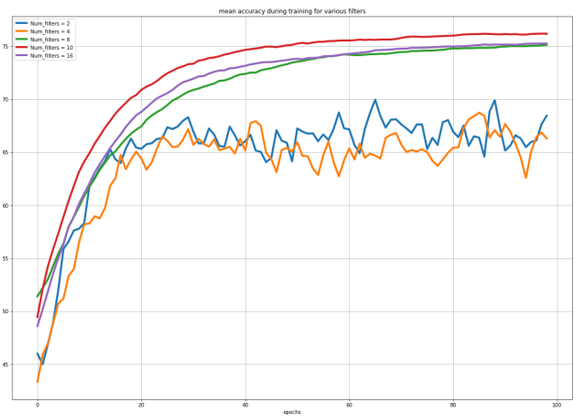


Figure 11: Accuracy of predictions with different values of filters

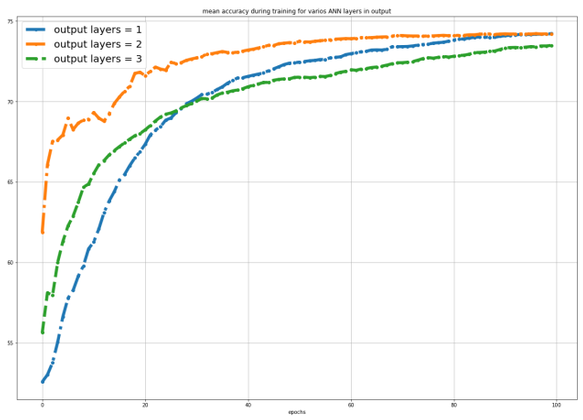


Figure 12: Accuracy of predictions with different values output layers

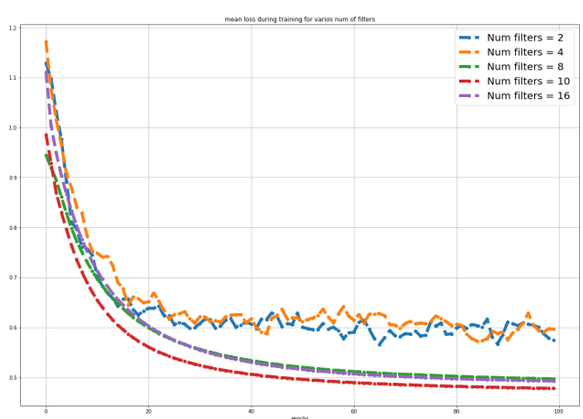


Figure 13: Loss evolution for different filters during training

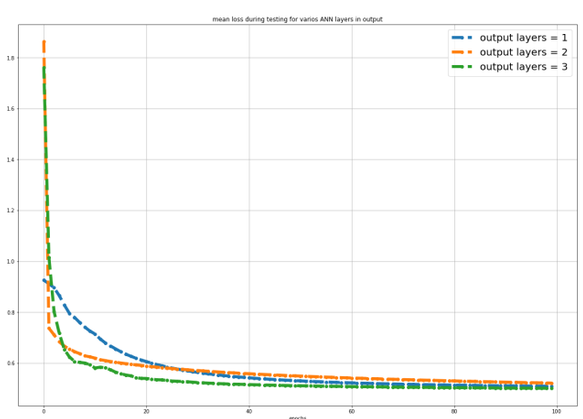


Figure 14: Loss evolution for different output layers during training

For the test set one architecture seems to perform better :8 filters and one fully connected layer as output. It keeps the accuracy of 74% for the test set, while the architecture with 2 layers seems to perform less for the testing. But i cannot jump to a quick conclusion, the model is still tunable : with better device (GPU instead of CPU) and better values for learning rate, and maybe some dropout, the model could do better. The results of testing are shown bellow :

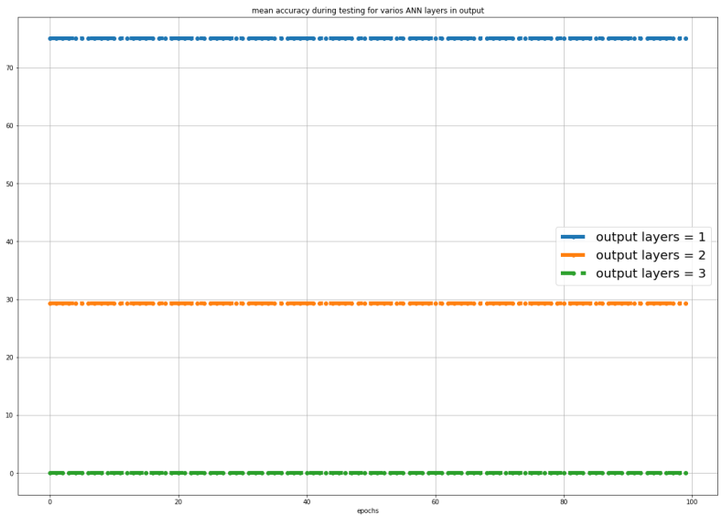


Figure 15: Accuracy during testing

And for timing, note that the models with 2, 4, 10, 16 filters respectively, take approximately about 4, 6, 10 and 14 hours to train. We used an intel core i7 8^{th} generation with 8 Cpu(S) with 1.80 GHz each without any GPU. Note also that random access memory is crucial during training.

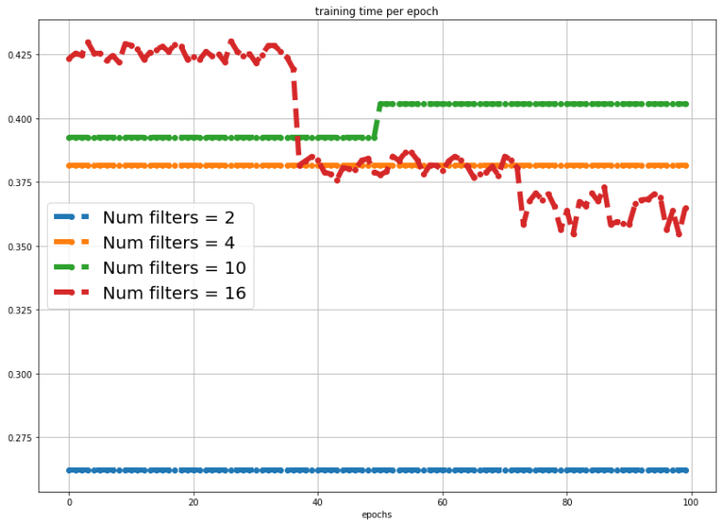


Figure 16: timing of training per epoch for different filters, values in minutes

7 Generalisation : case of $3D$ tensors

In the recent study, the architecture is valid for second order tensor. Many papers published some generalizations of the matrix product to the order 3 tensors. It is called the t-product. The generalizations even covers some classic decomposition such as SVD. It would be a great job to make a generalisation of the GrNet architecture to order 3 tensors with the t-product. Another step would be to utilize the **Canonical Polyadic decomposition** to tensors of order p, $(p > 3)$.

8 Conclusion : discussion and comments

The architecture has shown to be efficient for handling Grassmannian data inputs. we obtained the highest accuracy of 75% in the test set for one fully connected layer + softmax final output layer. In terms of timing, it's quit reasonable, as we didn't use any GPU. Although, there are some crucial points to discuss : mainly the learning slowdown problem, and the weights update procedure for filters $F = \{\mathbf{W}_1, \mathbf{W}_2, ..., \mathbf{W}_m\}$.

8.1 the learning slowdown problem

First let's see what the form of Softmax function ad its derivative looks like :

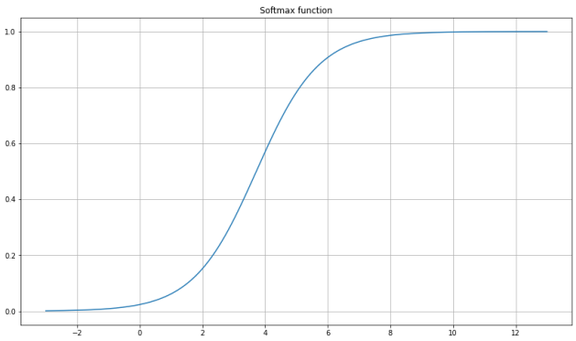


Figure 17: Softmax function plot

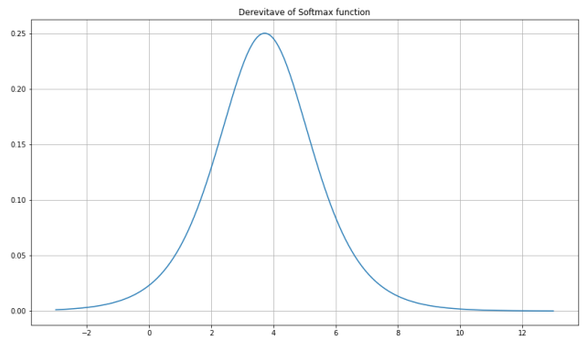


Figure 18: The derivative of Softmax

As we can see, when the Softmax outputs, get close to 1, the the evolution of the Softmax is quit slow, compared to half probability value 0.5 region. That means that the values of the derivative get close to 0 quickly and remain close to zero until infinity. This means during the training, when our network starts to predict high probability values, which means starts giving more more precise predictions, values of Softmax layer, become close to one, thus the evolution of quantities $\frac{\partial l}{\partial b}$ and $\frac{\partial l}{\partial W}$ becomes more and more slow, and **this explains that the accuracy curve starts to be flattened near** 75%.

One solution to this, is to add a discriminating term in the loss. This is called the **regularization** technique. Michael Nielsen in [8] adds the term $\lambda \|\mathbf{W}\|_F$, where $\lambda > 0$ and $\|.\|_F$ the Frobenius norm. This changes equation 26 giving the term $\frac{\partial l}{\partial w_{a,b}}$, and the SGD in equation 28. Please refer to his outstanding book[8] for more details.

Note also that the learning procedure, briefly discussed in 4.2.2 is the default version of SGD. But as the update of filters occurs in the manifold of fixed rank matrices, the update procedure for the filters differs from that one of bias $\mathbf{b}$ and FC weights $\mathbf{W}$, as for the filters we perform a projected SGD. The formula of updating the filters is :

$$\mathbf{W} = \mathbf{r}(\ \mathbf{W} - \alpha \text{ReimGrad}\ ),\ \forall\ \mathbf{W} \in \mathbf{F} = \{\mathbf{W}_1, \mathbf{W}_2, ..., \mathbf{W}_m\}. \tag{29}$$

Here function $\mathbf{r}$ denotes a retraction, ie the projection onto the manifold of fixed rank matrices, as described in [7] : ($\sigma_i \mathbf{u}_i \mathbf{v}_i$ obtained with SVD decomposition of $\mathbf{X}$)

$$\mathbf{r}(\mathbf{X}) = \sum_{i=1}^{\text{rank}(\mathbf{X})} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{30}$$

8.2 the update procedure and explosion of weights

When computing the Reimannian gradient from euclidean gradient, the original paper propose the following algorithm :

```python
import numpy as np
def call_Reimann_grad(W, EucGrad):
    """
    W : weight to be updated
    EucGrad : euclidean gradient
    """
    EucGradT = EucGrad.astype(np.double).transpose()
    W = W.astype(np.double)
    U, _, V = np.linalg.svd(np.dot(W, EucGradT))
    Q = np.dot(V, U.transpose())
    Rgrad = np.dot(EucGradT, Q) - W.transpose()
    return Rgrad.astype(np.double)
```

Listing 2: Reimannian grad from euclidean grad

```
Traceback (most recent call last):

  File "/home/mohammedhssein/Documents/stage/grnet/model_train.py", line 59, in <module>
    logits = model(input)

  File "/home/mohammedhssein/anaconda3/lib/python3.7/site-packages/torch/nn/modules/module.py", line 541, in __call__
    result = self.forward(*input, **kwargs)

  File "/home/mohammedhssein/Documents/stage/grnet/grnet_model.py", line 61, in forward
    X5 = utils.call_orthmap(X4)

  File "/home/mohammedhssein/Documents/stage/grnet/utils_model.py", line 122, in call_orthmap
    return OrthMapLayer().apply(input)

  File "/home/mohammedhssein/Documents/stage/grnet/utils_model.py", line 75, in forward
    U, Sig, Ut = torch.svd(input[i, :, :])

RuntimeError: svd_cpu: the updating process of SBDSDC did not converge (error: 11)
```

Figure 19: SVD doesn't converge : weights increased exponentially

As we see, the algorithm return the gradient without dividing it with the Frobenius norm. With this technique, the learning process stops at epoch 10. The weights explode : values are near 10^{30} and -10^{30}. To ensure the weights still in between 0 and 1, we divided by the norm. Therefore, the update procedure is :

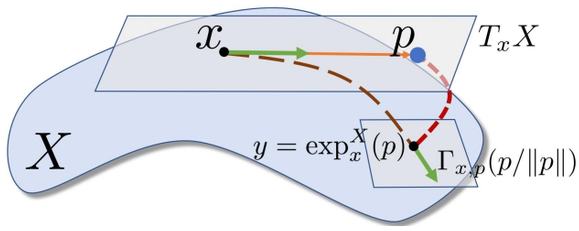


Figure 20: retraction function

```python
def update_params_model_v2(W, EucGrad, lr):
    '''
    Update procedure : projected SGD
    '''
    reim_grad = call_Reimann_grad(W, EucGrad)
    reim_grad = reim_grad.transpose()
    to_map = (W - lr*reim_grad)/np.linalg.norm(W - lr*reim_grad)
    w = retraction(to_map)
    return w
```

Listing 3: Retraction function and update procedure

Many questions are still to check : is there a way to ensure filters matrices F remain in a **bounded manifold** in addition of being row full rank, for instance **orthogonal matrices** ?

References

[1] Jiayao Zhang, Guangxu Zhu, Robert W. Heath Jr., and Kaibin Huang, "**Grassmannian Learning: Embedding Geometry Awareness in Shallow and Deep Learning**", https://arxiv.org/pdf/1808.02229.pdf

[2] Zhiwu Huang , Jiqing Wu, Luc Van Gool "**Building Deep Networks on Grassmann Manifolds**", https://arxiv.org/pdf/1611.05742.pdf

[3] Zhiwu Huang , Jiqing Wu, Luc Van Gool "**GitHub repository for building Deep Networks on Grassmann Manifolds**", https://github.com/zhiwu-huang/GrNet

[4] Catalin Ionescu , Orestis Vantzos , and Cristian Sminchisescu "**Training Deep Networks with Structured Layers by Matrix Backpropagation**", https://arxiv.org/pdf/1509.07838.pdf

[5] PyTorch documentation, https://pytorch.org/docs/stable/index.html

[6] Understanding Principal Componenet Analysis http://ethen8181.github.io/machine-learning/dim_reduct/PCA.html

[7] Bart Vandereycken, Low-rank matrix completion by Riemannian optimization, https://arxiv.org/pdf/1209.3834.pdf

[8] Michael Nielsen, Neural networks and deep learning http://neuralnetworksanddeeplearning.com/chap3.html