UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica e dell'Automazione

METODI DI APPRENDIMENTO BASATI SUL GRADIENTE ESTESI AL GRASSMANN MANIFOLD APPLICATI AL CLUSTERING AUTOMATIZZATO

GRADIENT-BASED LEARNING METHODS EXTENDED TO THE GRASSMANN MANIFOLD APPLIED TO AUTOMATED CLUSTERING

Relatore:
Chiar.mo Prof.
SIMONE FIORI

Presentata da:
ALKIS KOUDOUNAS

Correlatore:
Chiar.mo Prof.
TOSHIHISA TANAKA

Sessione Autunnale: 1° Appello
Anno Accademico 2018/19

Contents

List of Figures

List of Tables

Sommario

Questa tesi ha l'obiettivo di estendere su manifold gli algoritmi di machine-learning classici e moderni basati sul gradiente, con lo scopo di consentire ad un algoritmo adattativo di apprendere in maniera ottimale una configurazione interna vincolata. La ragione che ha spinto alla stesura del seguente lavoro è stata quella di migliorare le prestazioni di una tecnica di clustering "sparso", di recente introduzione, basata sulla rappresentazione su manifold Grassmanniano, col fine di applicarla ad un dataset di immagini di grandi dimensioni. I risultati ottenuti confermano che gli algoritmi di apprendimento proposti, basati sul calcolo su manifold, risultano essere meno onerosi dal punto di vista della complessità computazionale rispetto a quelli già esistenti, pur non andando a compromettere l'efficienza del clustering.

Abstract

The purpose of the present thesis is to extend classical and modern gradient-based machine-learning algorithms to smooth manifolds in order to enable a machine to learn an optimal constrained internal configuration. The motivation of the present endeavour was to improve the performances of a recently-introduced sparse clustering technique based on Grassmann manifold representation to be applied to large-size pictorial datasets. The obtained results confirm that the proposed learning algorithms, based on manifold calculus, proved lighter in computational complexity than existing ones without detriment in clustering efficacy.

Chapter 1

Introduction

Optimization occurs spontaneously in nature. The motion of a free body that falls towards the surface of the earth is subject to at least two optimization principles: the potential energy on the surface of the earth is minimal, hence the body seeks to optimize its height according to the gravitational potential; in addition, the free body falls along a straight line, which is the shortest path among all possible paths connecting the initial location to the center of the earth, hence the body seeks to optimize the length of its journey. Likewise, a metallic pellet constrained to slide over the surface of a sphere, whose north pole be magnetized, will move toward the magnetized pole, in order to optimize the magnetic potential, along the shortest path connecting the current location to the north pole, that is an arc of the great circle passing through these points.

Gradient-based optimization methods stay at the very core of algorithms that afford an artificial system to learn and improve its performances and are invoked whenever a system's performances are evaluated through a smooth criterion function. A criterion function affords the evaluation of any given configuration of the parameters of an artificial learning system and the purpose of the optimization method is to seek for the best configuration of parameters that minimizes the discrepancy between the current system's performances and the expected performances.

The behavior of a gradient-based optimization method depends on the shape of the criterion surface as, for instance, how 'deep' is a local minimum or how far are local minima from one another. Starting from the basic gradient-steepest descent method, which seeks local extrema of a criterion function by pursuing the direction indicated by the function's gradient, a number of gradient-based methods were derived. Each method in this category was developed to fix a specific issue arising in a specific situation. In the Chapter 2 of this paper, we are going to revise a number of classical and modern gradient-based learning algorithm, such as the basic gradient descent algorithm in Section 2.1, the stochastic gradient decent in Section 2.2, the mini-batch stochastic gradient descent algorithm in Section 2.3, the gradient descent with momentum in Section 2.4, the Nesterov accelerated gradient based algorithm in Section 2.5, the adaptive gradient method in Section 2.6, the AdaDelta algorithm in the Section 2.7 and the adaptive moment estimation method in Section 2.8.

In several cases of interest, the parameters of a learning system are independent from one another, therefore the search space is $\mathbb{R}^n$, where the dimension n of the search space might be large (this is the case, for example, of a multilayer perceptron endowed with several layers and several neurons per layer). Over recent years, it occurred to researchers in this area that the parameters of a learning system may be subjected to mutual, non-linear (even very involved) constraints. If the constraints are smooth and holonomic, the constraints themselves might be represented by a smooth manifold $\mathbb{M} \subset \mathbb{R}^n$. In this event, the optimization methods at the core of systems' learning procedures need to be reformulated in terms of manifold language, namely, manifold calculus and numerical analysis on manifolds.

The basic gradient steepest descent learning method on manifold is already available from the scientific literature and found widespread application in machine learning (see, for example, [2, 6]). Since basic gradient descent suffers of known drawbacks, we endeavoured to extend a number of classical and modern gradient-based learning methods to a general smooth manifold, as illustrated in the Chapter 3. As a special case of particular interest in the present work, we recalled some definitions and details about the Grassmann manifold in the Section 3.8.

In several applications – such as machine learning, image processing and computer vision – high dimensional data are widespread [13]. Grassmann manifolds are abstract manifolds whose elements are subspaces. As such, Grassmann manifolds are natural candidates for data reduction and sparse representation, a necessary step in classification by high-dimensional data clustering.

Clustering is one of the most widely used data exploration tools. Its goal is to partition data points into several groups such that points in the same group are similar to one another, according to a pre-defined similarity measure, and points in different groups are dissimilar from each other. To this aim, the main steps to take are creating a similarity/affinity matrix for a given dataset, and performing clustering to categorize data samples.

These two major steps determine the performance of spectral clustering methods. The goal of *Spectral (or Subspace) Clustering* (SC) [9, 11], which is a simple extension of traditional clustering, is to cluster data points that lie in a union of low-dimensional subspaces, while the key idea of *Sparse Spectral Clustering* (SSC) is that, among the infinitely many possible representations of a data point in terms of other points, a sparse representation corresponds to selecting a few points from the same subspace. This motivates solving a sparse optimization program whose solution is used in a spectral clustering framework to infer the clustering of the data into subspaces. The objective of SSC itself is to characterize how close the eigen-structure of a similarity/affinity matrix is to a partition implied by the latent representation. Put in another way, instead of explicitly inferring latent representation for eigen-structure, learning the subspace structure should be desired. This proves that it is necessary and more reasonable to implement the SSC optimization over subspaces, – such are the points on Grassmann manifolds. GSC [15] algorithm introduces a straightforward way to optimize the sparse clustering objective introduced in [9] by adopting Grassmann manifold optimization strategy, in order to learn a better and efficient latent feature representation.

The primary contributions of this thesis are:

1. We follow the GSC algorithm proposed in [15] and we study a computationally more efficient and faster way to compute (4.13);

2. We extend a number of classical and modern gradient-based learning methods to a general smooth manifold;

3. We evaluate the performance of the GSC algorithm learnt by these gradient-based learning methods via both the clustering on toy datasets and real-world databases.

This thesis is organized as follows. In Chapter 2, we summarize a number of classical and modern gradient-based learning algorithm, while Chapter 3 focuses on the extension of optimization algorithms to smooth manifolds, after briefly recalling manifold notation. The Chapter 4 of this paper contains a review of clustering in machine learning, with particular emphasis on the basic aspects of the SSC and the GSC algorithms. In this chapter, the main steps of the two most important clustering methods – that are *k-means*++ and *NCut* – are also recalled. In Chapter 5, the performance of the GSC algorithm learnt by all the gradient-based learning methods is assessed via both the clustering on synthetic and pictorial real-world datasets. Chapter 6 concludes the paper.

Chapter 2

Summary of optimization methods in $\mathbb{R}^n$

The present chapter outlines a number of classical as well as modern learning schemes based on parameter optimization known from the machine learning literature.

2.1 Gradient descent (GD)

Gradient descent is an optimization algorithm used to minimize a given convex function (the so-called *loss function* $J(\theta)$, where θ is a parameter vector in $\mathbb{R}^n$) to one of its local minima by iteratively moving in the direction of steepest descent as defined by the opposite direction of the gradient.

The very first thing to do is defining the initial parameters values θ_0, and from there on *Gradient descent* iteratively adjusts the values, using calculus, so that they minimize the given cost-function. How big the steps are that *GD* takes into the direction of the local minimum are determined by the *learning rate* η (usually $\eta \in (0,1]$): it determines how fast or slow the movement towards the optimal weights will be. In formulas we have:

$$\theta_{t+1} = \theta_t - \eta \, \nabla J(\theta_t), \tag{2.1}$$

where $\nabla J(\theta)$ denotes the gradient of the criterion function J evaluated at a point θ and $t = 0,\ 1,\ 2,\ \ldots$ denotes an iteration index.

In order for *GD* to reach the local minimum, the *learning rate* has to be set to an appropriate value, which is neither too low nor too high. This is because if the steps are too big, it maybe will not reach the local minimum because it just bounces back and forth between the convex function of *GD*. If the *learning rate* is set instead to a very small value, *GD* will eventually reach the local minimum but it will maybe take too much time. This is the reason why the *learning rate* should be neither too high nor too low.

2.2 Stochastic gradient descent (SGD)

Stochastic Gradient Descent is an optimization algorithm which improves the efficiency of the *Gradient Descent* algorithm. It performs a series of steps to minimize the given cost function, but unlike *GD*, which is computationally expensive to run on large data sets since the cost gradient's computation is based on the complete training set, *SGD* is able to take smaller steps - the smaller the step, the more frequent the update of the parameters - to be more efficient while achieving the same result.

The Algorithm 1 shows how this optimization method works.

Algorithm 1 Stochastic Gradient Descent (SGD)

```
1: Randomly sample a point $i$.
2: Update the parameters by
       $$\theta_{t+1} = \theta_t - \eta \, \nabla J(\theta_t; x^{(i)}; y^{(i)}). \quad (2.2)$$
3: Return to 1.
```

2.3 Mini-batch stochastic gradient descent

Mini-batch Stochastic Gradient Descent is a trade-off between *Stochastic Gradient Descent* and *Gradient Descent*. In *Mini-batch Stochastic Gradient Descent*, the cost function (and therefore gradient) is averaged over a small number of samples $m \ll n$ (the number of samples m is usually selected through a random permutation). This is opposed to the *SGD* batch size of 1 sample, and the *GD* size of all the training samples. This way the algorithm is much faster, even if there are still some unsolved problems. First of all the choice of the *learning rate* η, which should be neither too high nor too low, in order to avoid the same problems we faced in the *GD* optimization. Another notable limitation is that *Mini-batch Stochastic Gradient Descent* algorithm tends to get stuck in local minima.

2.4 Gradient descent with momentum

SGD has trouble navigating areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. The *Momentum* is a method that helps accelerate *SGD* along the relevant directions and softens the oscillations in the irrelevant directions [12]. It does this by simply adding a fraction γ (usually $\gamma \in (0,1]$) of the direction of the previous step to a current step: this achieves amplification of speed in the correct direction and mellows oscillation in wrong directions, so that the *Gradient Descent* step could be larger, compared to *SGD*'s constant step. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, faster convergence and reduced oscillation are gained. In formulas:

$$\begin{cases} v_t = \gamma v_{t-1} + \eta \, \nabla J(\theta_t), \\ \theta_{t+1} = \theta_t - v_t. \end{cases} \tag{2.3}$$

However, one of the most troubling problem is that *Momentum* tends to miss or oscillate around the minima.

2.5 Nesterov accelerated gradient (NAG)

Nesterov Accelerated Gradient is a simple change to normal *momentum* and it overcomes its main problem by starting to slow down early [10]. For a recent review see, e.g. [3].

In this algorithm indeed the gradient term is not computed from the current position θ_t in parameter space, but instead from an approximated new “look-ahead” position $\hat{\theta}_t = \theta_t - \gamma v_{t-1}$. This helps because while the gradient term always points in the right direction, the momentum term may not. If the momentum term points in the wrong direction or overshoots, the gradient can still “go back” and correct it in the same update step. So, *NAG* basically exploits that knowledge, and instead of using the current position's gradient, it uses the next approximated position's gradient with the assumption that it will give better information before taking the next step. The revised parameter updating rule is:

$$\begin{cases} \hat{\theta}_t := \theta_t - \gamma v_{t-1} \\ v_t = \gamma v_{t-1} + \eta \, \nabla J(\hat{\theta}_t), \\ \theta_{t+1} = \theta_t - v_t. \end{cases} \tag{2.4}$$

2.6 Adaptive gradient (AdaGrad)

AdaGrad was invented trying to solve the problem with *learning rate* in *GD*, which is that the *learning rate* is constant and affecting all the parameters [4]. In order to do so, *AdaGrad* allows the *learning rate* to adapt based on parameters. It performs larger updates for infrequent parameters and smaller updates for frequent ones. Because of this, it is well suited for sparse data (Natural Language Processing (NLP) or Image Recognition). Another advantage is that it basically eliminates the need to tune the *learning rate*. Each parameter has its own *learning rate* and, due to the peculiarities of the algorithm, the *learning rate* is monotonically decreasing. This causes however the biggest problem: at some point of time the *learning rate* is so small that the system stops learning.

What *AdaGrad* actually does is to accumulate the sum of squared of all the parameters' gradient, and use that to normalize the learning rate η, so that now η could be smaller or larger depending on how the past gradients behaved: parameters that updated a lot will be slowed down while parameters that received little updates will be have bigger learning rate to accelerate the learning process.

The revised parameter update rule for the *AdaGrad* algorithm is:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} \nabla J(\theta_t)_i \tag{2.5}$$

$G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each element in the diagonal (i,i) is the sum of squared gradient estimate over the course of training, up to the *time-step t*. In formulas:

$$G_{t,ii} = \sum_{\tau=1}^{t} (\nabla J(\theta_\tau)_i)^2. \tag{2.6}$$

Note that the parameters update is point-wise operation, hence the learning rate is adaptive per-parameter. Furthermore, the ε is useful to avoid the division by zero, so that the optimization becomes numerically stable: that's why it's usually set with considerably small value, like 10^{-8}.

2.7 AdaDelta

AdaDelta resolves the continually decaying *learning rate* in *AdaGrad* by using a limited sliding window (which allows the sum to decrease), instead of summing all past square roots, although the actual accumulation process is implemented using a concept from *Momentum* [16].

The highlights of *AdaDelta* algorithm are summarized in the Algorithm 2.

The *AdaDelta* learning algorithm attempts to alleviate the task of choosing a *learning rate* by introducing a new dynamic *learning rate* that is computed on a per-dimension basis using only first order information.

Algorithm 2 AdaDelta

```
1: Compute gradient $\nabla J(\theta)$ at the current time $t$
2: Accumulate gradient
       $E[\nabla J(\theta)_i^2]_t = \gamma E[\nabla J(\theta)_i^2]_{t-1} + (1-\gamma)\nabla J(\theta_t)_i^2.$    (2.7)
3: Compute update
       $(\Delta\theta_t)_i = -\frac{\sqrt{E[(\Delta\theta)_i^2]_{t-1}+\varepsilon}}{\sqrt{E[\nabla J(\theta)_i^2]_t+\varepsilon}} \cdot \nabla J(\theta_t)_i.$    (2.8)
4: Accumulate updates
       $E[(\Delta\theta)_i^2]_t = \gamma E[(\Delta\theta)_i^2]_{t-1} + (1-\gamma)(\Delta\theta_t)_i^2,$    (2.9)
   where $\gamma$ is a decay constant similar to that used in the *Momentum* method.
5: Apply update
       $(\theta_{t+1})_i = (\theta_t)_i + (\Delta\theta_t)_i.$    (2.10)
```

2.8 Adaptive Moment Estimation (AdaM)

Similar to *AdaDelta*, *AdaM* is an algorithm computationally efficient, little-memory demanding and also appropriate for non-stationary objectives [8]. This method computes individual adaptive *learning rates* for different parameters from estimates of first and second moments of the gradients. The algorithm updates exponential moving averages of the gradient m_t and the squared gradient v_t where the hyper-parameters $\beta_1, \beta_2 \in [0,1)$ control the exponential decay rates of these moving averages. The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient. However, these moving averages are initialized as (vectors of) 0's, leading to moment estimates that are biased towards zero, especially during the initial timesteps, and especially when the decay rates are small. The good news is that this initialization bias can be easily counteracted, resulting in bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$.

Hence, the main steps of this method are shown in the Algorithm 3.

The authors of the algorithm proposed default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε. They show empirically that *AdaM* algorithm works well in practice and compares favorably to other adaptive learning-method algorithms.

Algorithm 3 Adaptive Moment Estimation (AdaM)

```
1: Compute gradient $\nabla J(\theta)$ at the current time $t$
2: Update biased first moment estimate
       $$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_t). \quad (2.11)$$
3: Update biased second raw moment estimate
       $$v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2)(\nabla J(\theta_t)_i)^2. \quad (2.12)$$
4: Compute bias-corrected first moment estimate
       $$\hat{m}_t = \frac{m_t}{1-\beta_1^t}. \quad (2.13)$$
5: Compute bias-corrected second raw moment estimate
       $$\hat{v}_t = \frac{v_t}{1-\beta_2^t}. \quad (2.14)$$
6: Update parameters
       $$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\varepsilon}\hat{m}_t. \quad (2.15)$$
```

Chapter 3

Extension of optimization algorithms to smooth manifolds

The present chapter aims at extending the optimization algorithms recalled in the previous section to Riemannian manifolds. The reader should keep in mind that extending an optimization algorithm from a flat space like $\mathbb{R}^n$ to a Riemannian manifold $\mathbb{M}$ is neither straightforward nor univocal.

3.1 Manifold notation

Once we have understood and verified the functioning of these algorithms on $\mathbb{R}^n$, we want to use them to optimize functions on a manifold $\mathbb{M}$.

"The general notion of manifold $\mathbb{M}$ is quite difficult to define precisely. A surface gives the idea of a two-dimensional manifold. If we take for instance a sphere or a torus, we can decompose this surface into a finite number of parts such that each of them can be bijectively mapped into a simply-connected region of the Euclidean plane". This is the beginning of the third chapter of *Leçons sur la Géométrie des espaces de Riemann* by Elie Cartan. He explains that these parts are that we call *open sets*, and he also describes that if the domains of definition of two such maps (which are now called *charts*) overlap, one of them is gotten from the other by composition with a smooth map of the Euclidean space. This is just the formal definition of a differential (or smooth) manifold.

A n-dimensional manifold can be however informally defined as a set $\mathbb{M}$ covered with a *suitable* collection of coordinate patches, or charts, that identify certain subsets of $\mathbb{M}$ with open subsets of $\mathbb{R}^n$. Such a collection of coordinate charts can be thought of as the basic structure required to do differential calculus on $\mathbb{M}$.

In order to be able to use these optimization algorithms on Manifold, we have to introduce some basic concepts first. Let $\mathbb{M}$ be a real differentiable manifold of dimension n. At a point $U \in \mathbb{M}$, the tangent space to the manifold $\mathbb{M}$ is denoted $T_U\mathbb{M}$. The symbol $T\mathbb{M}$ denotes the tangent bundle defined as $T\mathbb{M} = \{(U,V) \mid U \in \mathbb{M}, V \in T_U\mathbb{M}\}$. Knowing that, it is obvious that each point of $\mathbb{M}$ has its own tangent bundle that is different from one point to another. Hence, in Manifold geometry, we can recognize only two things: points belonging to the Manifold and vectors belonging to a tangent space. Once this point was resolved, we can introduce two basic operators: the *exponential map* and the *parallel translation*.

The *exponential map* is a map from a subset of a tangent space $T_U\mathbb{M}$ of a manifold $\mathbb{M}$ to $\mathbb{M}$ itself. Given a point $U \in \mathbb{M}$ and a vector $V \in T_U\mathbb{M}$, there is a unique geodesic[1] γ_V satisfying $\gamma_V(0) = U$ with initial tangent vector $\gamma_V'(0) = V$. The corresponding *exponential map* is defined by $\exp_U(V) = \gamma_V(1)$.

The second operator that we need to define is the *parallel translation*. It takes a point $U,V \in T\mathbb{M}$ and a vector $W \in T_U\mathbb{M}$ as input and transports the vector W along a geodesic arc departing from U along the direction V for a unit time. We will use the notation $\mathrm{P}_{U,V}(W)$. In the design of a numerical algorithm in Subsection 3.2, we shall invoke a version of parallel transport denoted by $\mathrm{P}_{U,V}(V)$, namely, the transport of a tangent vector along the geodesic line directed along itself. This concept exploits the well-known, defining property of a geodesic line to self-transport its own tangents.

Parallel translation is also an isometry, which means that the parallel transport changes the direction of a transported vector to make it conform to the geometry of the underlying manifold, without altering the length of the transported tangent vector[2].

In addition, we shall make us of the operator $\Pi_U[\cdot]$, which denotes an orthogonal projection over the tangent space $T_U\mathbb{M}$.

3.2 Gradient descent (GD) on $\mathbb{M}$

As mentioned earlier on $\mathbb{R}^n$, this method imposes to add to the variable that should be optimized a small fraction of the anti-gradient, so as to follow the right path. But unlike $\mathbb{R}^n$ where both addends were vectors, on *Manifold* a point on $\mathbb{M}$ has to be added to a vector of the respective tangent space. However this operation cannot be conducted directly, since it is not possible to add them without using the *exponential map*.

[1]A geodesic is a curve representing the shortest path between two points in a manifold.
[2]Each vector space on $\mathbb{M}$ has its own norm; therefore the norm has to be defined for each of them.

Hence, the update parameters operation follows this formula:

$$U_{t+1} = \exp_{U_t}(-\eta \nabla_{U_t} J), \tag{3.1}$$

where $U_t \in \mathbb{M}$, $\eta \in \mathbb{R}^+$ and $\nabla_{U_t} J \in T_{U_t}\mathbb{M}$, since it represents the Riemannian gradient of the cost function J calculated in the point U_t. To start the iteration, it is necessary to choose an initial guess U_0. The iteration step runs over $t = 0,\ 2,\ 3,\ \dots,\ K$, where K denotes a pre-defined number of iterations.

This algorithm, as well as all the following ones, has been implemented in such a way that the iteration loop ends once a predetermined number of iterations is reached, number which depends on the complexity of the initial data and on the algorithm itself.

3.3 Gradient descent with momentum on $\mathbb{M}$

Faithful to the original idea, instead of using only the gradient of the current step to guide the search, *Momentum* also accumulates the gradient of the past steps to determine the direction to go. Since gradients are calculated at different points, they belong to different tangent bundles: $\nabla_{U_{t-1}} J \in T_{U_{t-1}}\mathbb{M}$ whereas $\nabla_{U_t} J \in T_{U_t}\mathbb{M}$, therefore it is not possible to add these terms directly. *Parallel translation* has to be used to transport $\nabla_{U_{t-1}} J$ from $T_{U_{t-1}}\mathbb{M}$ to $T_{U_t}\mathbb{M}$, as as it can be added to $\nabla_{U_t} J$, since they now both belong to the same tangent space. The parameters are then updated through the *exponential map*.

The formulas below show the essential steps of this extended learning algorithm:

$$\begin{cases} V_t = \gamma \mathrm{P}_{U_{t-1},V_{t-1}}(V_{t-1}) + \eta\, \nabla_{U_t} J, \\ U_{t+1} = \exp_{U_t}(-V_t) \end{cases} \tag{3.2}$$

where $U_t \in \mathbb{M}$, $\eta > 0$ denotes a learning stepsize, $\gamma > 0$ is a momentum coefficient, and $\nabla_{U_t} J, V_t \in T_{U_t}\mathbb{M}$. The iteration step runs over $t = 1,\ 2,\ 3,\ \dots,\ K$, where K denotes a pre-defined number of iterations. The initial point U_0 is chosen in $\mathbb{M}$ and the initial velocity V_0 may be either randomly picked in $T_{U_0}\mathbb{M}$, or set to $\nabla_{U_0} J$, or set to zero.

3.4 Nesterov accelerated gradient (NAG) on $\mathbb{M}$

The idea behind *NAG* is that instead of calculating the gradient at the current position, it calculates the gradient at the position where the momentum is about to arrive, called as “look-ahead” position, and by that time a fraction of this gradient is added to the previous ones. Since gradients are calculated at different points, they belong to different tangent spaces here too, therefore they must be taken back to the same tangent space through the projection operator before updating the parameters. In formulas, this is written as:

$$\begin{cases} \hat{U}_t = \exp_{U_t}(-\gamma \Pi_{U_t}(V_{t-1})), \\ V_t = \Pi_{U_t}\left(\gamma V_{t-1} + \eta \nabla_{\hat{U}_t} J\right), \\ U_{t+1} = \exp_{U_t}(-V_t), \end{cases} \tag{3.3}$$

where $U_t \in \mathbb{M}$, $\eta > 0$ denotes a learning stepsize, $\gamma > 0$ denotes a forgetting factor and $\nabla_{\hat{U}_t} J, V_t \in T_{U_t}\mathbb{M}$. The iteration step runs over $t = 1, 2, 3, \ldots, K$, where K denotes a pre-defined number of iterations. The initial point U_0 is chosen in $\mathbb{M}$ and the initial velocity V_0 may be either randomly picked in $T_{U_0}\mathbb{M}$, or set to $\nabla_{U_0} J$, or set to zero.

3.5 Adaptive gradient (AdaGrad) on $\mathbb{M}$

A direct extension of the original AdaGrad method would need a decomposition of the Riemannian gradient of the criterion function into components in order to weight any component according to the square root of the accumulated component squares. In formulas, if we denoted by $\{\partial_1, \partial_2, \ldots, \partial_d\}$ the canonical basis of the tangent space $T_{U_t}\mathbb{M}$, the Riemannian gradient $\nabla_{U_t} J$ may be decomposed as $\nabla_{U_t} J = \sum_{i=1}^{d} (\nabla_{U_t} J)_i \partial_i$, where each $(\nabla_{U_t} J)_i \in \mathbb{R}$ denotes one of the components of the gradient with respect to the canonical basis. The accumulated squared component may be updated as

$$G_{t+1,ii} = G_{t,ii} + (\nabla_{U_t} J)_i^2, \tag{3.4}$$

and a *normalized gradient* may be defined as follows:

$$\tilde{\nabla}_{U_t} J := \sum_{i=1}^{d} \frac{(\nabla_{U_t} J)_i}{\sqrt{G_{t,ii} + \varepsilon}} \partial_i. \tag{3.5}$$

Although $\tilde{\nabla}_{U_t} J$ does no longer represent a Riemannian gradient of the criterion function J, it is still a tangent vector in $T_{U_t}\mathbb{M}$, therefore it is mathematically sound to update the current point U_t to the next point by

$$U_{t+1} = \exp_{U_t}(-\eta \tilde{\nabla}_{U_t} J). \tag{3.6}$$

The set of equations (3.4) and (3.5) are faithful to the original concept but are quite impractical due to the need of getting back and forth to the component representation and due to the need of calculating the canonical basis of each tangent space encountered during the optimization process.

Assuming that the manifold $\mathbb{M}$ is a matrix manifold (or that its elements may be represented as matrices), a possible workaround consists in weighting every single entry of the gradient by a weight that is inversely proportional to the square root of the accumulated square of the same entry across time. In formulas, a possible workaround may be expressed as follows:

$$\begin{cases} G_t = G_{t-1} + \nabla_{U_t} J \odot \nabla_{U_t} J, \\ \hat{G}_t = \Pi_{U_t}\left[\frac{\eta}{\sqrt[\circ]{G_t + \varepsilon}} \odot \nabla_{U_t} J\right], \\ U_{t+1} = \exp_{U_t}(-\hat{G}_t), \end{cases} \tag{3.7}$$

where $\odot$ denotes the Hadamard (component-wise) matrix product and $\sqrt[\circ]{}$ denotes a component-wise square root. here, $\eta > 0$ denotes a learning stepsize and ε is a small-valued constant that prevents division by zero. Notice that both summation of a matrix by a constant and division between two matrices are intended component-wise. The iteration step runs over $t = 1,\ 2,\ 3,\ \ldots,\ K$, where K denotes a pre-defined number of iterations. The necessity of the projection operator is quite apparent since the result of the Hadamard product $\frac{\eta}{\sqrt[\circ]{G_t+\varepsilon}} \odot \nabla_{U_t} J$ between a weighting matrix and a Riemannian gradient apparently is not a tangent vector any longer.

3.6 AdaDelta on $\mathbb{M}$

Similarly to the AdaGrad method, the original AdaDelta algorithm may be extended to a Riemannian manifold (and, indeed, even to non-Riemannian smooth manifolds) in a number of ways. In the following, a mathematically sound version is proposed:

$$\begin{cases} S_t = \gamma S_{t-1} + (1-\gamma)\cdot(\nabla_{U_t} J \odot \nabla_{U_t} J), \\ \hat{G}_t = \Pi_{U_t}\left[\sqrt[\circ]{\frac{\Delta_{t-1}+\varepsilon}{S_t+\varepsilon}} \odot (\gamma \nabla_{U_t} J)\right], \\ \Delta_t = \gamma \Delta_{t-1} + (1-\gamma)\cdot(\hat{G}_t \odot \hat{G}_t), \\ U_{t+1} = \exp_{U_t}(-\hat{G}_t). \end{cases} \tag{3.8}$$

The first two equations are meant to provide the accumulated gradients and the accumulated updates as expected in the original AdaDelta method.

Notice that the two matrix-sequences Δ_t and S_t do not show any particular structure. Also, it is worth underlining that, in the equation that defines values of $\hat{G}$, the summation by a scalar as well as the division between matrices are meant to be effected component wise and that the Riemannian gradient was further scaled by γ to improve the numerical stability of the learning algorithm. To start the iteration, it is necessary to choose an initial guess U_0. Moreover, we set $S_0 := 0$ and $\Delta_0 := 0$. The iteration step runs over $t = 1,\ 2,\ 3,\ \ldots,\ K$, where K denotes a pre-defined number of iterations.

3.7 Adaptive gradient with momentum (AdaM) on $\mathbb{M}$

The AdaM algorithm outlined in the Section 2.8 may be extended to a smooth manifold by an appropriate reformulation of its constituting equations. The proposed extension goes as follows:

$$\begin{cases} m_t = \beta_1 \Pi_{U_t}(m_{t-1}) + (1-\beta_1) \cdot \nabla_{U_t} J, \\ v_t = \beta_2 v_{t-1} + (1-\beta_2) \cdot (\nabla_{U_t} J \odot \nabla_{U_t} J), \\ \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \\ \hat{G}_t = \Pi_{U_t}\left[\frac{\eta}{\sqrt[\circ]{\hat{v}_t}+\varepsilon} \odot \hat{m}_t\right], \\ U_{t+1} = \exp_{U_t}(-\hat{G}_t), \end{cases} \tag{3.9}$$

where the notations β_1^t and β_2^t denote t-th powers.

Notice that the two matrix-sequences m_t and v_t do not show any particular structure. Also, it is worth underlining that, in the equation that defines values of $\hat{G}$, the summation by a scalar as well as the division between matrices are meant to be effected component wise. To start the iteration, it is necessary to choose an initial guess U_0. Moreover, we set $m_0 := 0$ and $v_0 := 0$. The iteration step runs over $t = 1, 2, 3, \ldots, K$, where K denotes a pre-defined number of iterations.

3.8 Grassmann Manifold $\mathrm{Gr}(n, p)$

The aim of this section is to recall some basic concepts about the Grassmann manifold. As a further reference, readers might consult [7]. Let n be a positive integer and let p be a positive integer, not greater than n. The set of p-dimensional linear subspaces of $\mathbb{R}^n$ is called *Grassmann manifold*, denoted by $\mathrm{Gr}(n, p)$. An element on a Grassmann manifold is generally represented by an arbitrarily chosen $n \times p$ full-rank matrix U, whose column spans the corresponding subspace. Given the large arbitrariness in the choice of a basis to represent a subspace, a sensible choice is to restrict oneself to a orthonormal (Stiefel) matrix. The Grassmann manifold can be represented by a collection of such generator matrices. Mathematically, this may be written as:

$$\mathrm{Gr}(n, p) = \{\mathrm{span}(U) \mid U \in \mathbb{R}^{n\times p}, U^\top U = I_p\}.$$

This allows to represent the generic element of Grassmann Manifold as $\mathscr{U} := \{UR \mid R \in \mathrm{O}(p)\}$, where $\mathrm{O}(p)$ denotes the orthogonal group (the set of $p \times p$ orthogonal matrices). An implication of this observation is that each element of $\mathrm{Gr}(n, p)$ is an equivalence set.

This allows the Grassmann Manifold to be treated as a quotient space of the larger Stiefel Manifold $\mathrm{St}(n,p)$, which is the set of matrices of size $n \times p$ with orthonormal columns. Specifically, the Grassmann Manifold has the quotient manifold structure

$$\mathrm{Gr}(n,p) := \mathrm{St}(n,p)/\mathrm{O}(p). \tag{3.10}$$

Hence, while optimization is conceptually on the Grassmann Manifold $\mathrm{Gr}(n,p)$, it numerically allows to implement operations with concrete matrices – that are elements of $\mathrm{St}(n,p)$.

Besides, given two points on Grassmann Manifold $U, \hat{U}$ and a tangent vector $V \in T_U\mathrm{Gr}(n,p)$ the Grassmann geodesic can be written as

$$\gamma(t) = [UB \quad A] \begin{bmatrix} \cos(\Sigma t) \\ \sin(\Sigma t) \end{bmatrix} B^\top, \tag{3.11}$$

where $A\Sigma B^\top$ is the compact SVD of V. Then the *exponential map*, denoted as $\exp_U(V) : T_U\mathrm{Gr}(n,p) \to \mathrm{Gr}(n,p)$, can be defined as the computation of $\hat{U} = \gamma(1)$ using the origin U and the tangent vector V.

Given V and W tangent vectors to the Grassmann Manifold at U, a formula for *parallel translation* of W along geodesic in the direction V can instead be described as follows:

$$\mathrm{P}_V(W,t) = \left[[UB \quad A] \begin{bmatrix} -\sin(\Sigma t) \\ \cos(\Sigma t) \end{bmatrix} A^\top + (I - AA^\top) \right] W. \tag{3.12}$$

Moreover, when the elements of a Grassmann manifolds are represented through Stiefel matrices, the projection operator over tangent spaces takes the expression $\Pi_U[A] := (I - UU^\top)A$, with A being any matrix of consistent size. As it is immediate to verify, computation-wise the projection over a tangent space is much more economical than parallel transport.

With that sorted, it is now possible to extend the above gradient-based learning methods to the manifold $\mathrm{Gr}(n,p)$ as needed.

Chapter 4

Sparse spectral clustering by Grassmann manifold optimization

The aim of this chapter is to summarize the main concepts about the *Grassmann Manifold Optimization Assisted Spectral Clustering* (GSC) algorithm, which is a clustering method based on the *Sparse Spectral Clustering* (SSC) algorithm, used to cluster a collection of multi-subspace data using sparse representation techniques.

4.1 Review of clustering in machine learning

Before going on, we start with an introduction of clustering to briefly explain why it is so important to introduce sparse representation with high-dimensional data.

Clustering is one of the most widely used data exploration tools. Its goal is to divide the data points into several groups such that points in the same group are similar and points in different groups are dissimilar to each other. In order to do this, the main steps are two: (1) creating a similarity/affinity matrix for the given data sample set, and (2) performing general clustering methods to categorize data samples – such as *k-means* or *NCut*.

In many applications – such as machine learning, image processing and computer vision – high dimensional data are widespread. This has unpleasant affects on the computation time and memory requirements of algorithms that want to extract information. However, it has been shown that high dimensional data often lie close to low-dimensional structures corresponding to several classes or categories to which the data belong. The goal of *Spectral (or Subspace) Clustering* (SC) [11, 9], which is a simple extension of traditional clustering, is to cluster data points that lie in a union of low-dimensional subspaces.

Sparse Spectral Clustering (SSC) [5] actually went even further. The key idea of this algorithm is that, among the infinitely many possible representations of a data point in terms of other points, a sparse representation corresponds to selecting a few points from the same subspace. This motivates solving a sparse optimization program whose solution is used in a spectral clustering framework to infer the clustering of the data into subspaces. The underlying idea behind the algorithm is what the authors call the *self-expressiveness* property of the data, which states that each data point in a union of subspaces can be efficiently represented as a linear or affine combination of other points.

Sparse spectral clustering specifies a sparsity-induced penalty to learn more clusters favored latent representation. Indeed, the objective of SSC is to characterize how close the eigen-structure of a similarity/affinity matrix is to a partition implied by the latent representation. In other words, instead of explicitly inferring latent representation for eigen-structure, learning the subspace structure should be desired. This proves that it is necessary and more reasonable to implement the SSC optimization over subspaces, – such are the points on Grassmann manifolds.

4.2 Spectral Clustering

Assume we are given

$$X = [x_1, \dots, x_N] \in \mathbb{R}^{P \times N}, \tag{4.1}$$

where X is a set of N data-points to be clustered and P is the dimension of data. The purpose of clustering is to partition the dataset X into k clusters according to certain similarity criteria.

Spectral Clustering partitions these N points into k clusters as specified in the Algorithm 4.

The elements of $UU^\top$ represents the similarity (or affinity) between the latent representation of the original data. In the ideal scenarios, $UU^\top$ can be permuted to block diagonal structure, which is privileged as it improves clustering performance. The idea of inducing or enforcing sparsity is the basis of *Sparse Spectral Clustering*. The SSC tries to obtain a better representation U by solving the following sparsity-induced optimization:

$$\min_{U \in \mathbb{R}^{N \times p}} \left(\left\langle UU^\top, L \right\rangle + \beta \|UU^\top\|_1 \right) \text{ s.t. } U^\top U = I, \tag{4.5}$$

where β represents a weight whose value promotes or demotes the sparsity of the solution U. The elements in UU^T corresponding to the weak inter-cluster connections tends to be zero, while the ones corresponding to the strong intra-cluster connections will be

Algorithm 4 Spectral Clustering (SC)

```
1: Compute the $N \times N$ affinity matrix $W$ defined by
   $$W_{ij} = \begin{cases} \frac{e^{-\|x_i - x_j\|^2}}{2\sigma^2} & \text{if } i \neq j, \\ 0 & \text{otherwise,} \end{cases} \quad (4.2)$$
   where $\sigma$ controls size of neighborhood.
2: Compute the normalized graph Laplacian
   $$L := I - D^{-1/2} W D^{-1/2}, \quad (4.3)$$
   where $D$ is the diagonal matrix with each diagonal element $d_{ii} := \sum_{j=1}^{N} w_{ij}$.
3: Compute $U \in \mathbb{R}^{N \times p}$ by solving the following constrained problem:
   $$\min_{U \in \mathbb{R}^{N \times p}} \left\langle UU^\top, L \right\rangle \text{ s.t. } U^\top U = I. \quad (4.4)$$
4: Form $\hat{U} \in \mathbb{R}^{N \times k}$ by normalizing each row of $U$ to have unit Euclidean length
5: Treat each row of $\hat{U}$ as a point in $\mathbb{R}^k$, and cluster them into $k$ groups by $k$-means or
   any other algorithm.
```

kept. But the solution U of the problem (4.5) may not be the best one. hat's why GSC algorithm has been proposed in [15].

We could sum it up as follows. Consider problem (4.5). Denote the objective function by

$$f(U) = \left\langle UU^\top, L \right\rangle + \beta \|UU^\top\|_1, \tag{4.6}$$

where L is the normal Laplacian graph. The constraint condition in problem (4.5) defines the Stiefel Manifold which consists, as we said before, of all the orthogonal column matrices:

$$\mathrm{St}(n,p) = \{U \in \mathbb{R}^{N \times p} \mid U^\top U = I\}. \tag{4.7}$$

As a consequence, problem (4.5) is an unconstrained manifold optimization problem on the *Stiefel manifold* $\mathrm{St}(n,p)$. Due to the definition of the Grassmann Manifold as the quotient space of the Stiefel Manifold *Stiefel manifold* $\mathrm{St}(n,p)$, it is possible to re-form the problem on the Grassmann Manifold as follows:

$$\min_{U \in \mathrm{St}(n,p)} \left(\left\langle UU^\top, L \right\rangle + \beta \|UU^\top\|_1 \right), \tag{4.8}$$

where equation (4.8) is an unconstrained Grassmann Manifold optimization problem. The objective function (4.6) of the new optimization problem (4.8) is not differentiable at the location where elements of $UU^\top$ are zero. However in this case the authors of [15] assumed using the sub-differential.

Before going on, it is better to introduce several notations. For a matrix A of size $m \times n$, $\mathrm{vec}(A)$ is a mn-dimensional vector by stacking columns of A one by one, and $\mathrm{ivec}(\mathrm{vec}(A)) = A$ the inverse operation of vec. $A \otimes B$ is the Kronecker product of matrices A and B. The transform $T_{m,n}$ is a matrix of size $mn \times mn$ such that $\mathrm{vec}(A) = T_{m,n}\mathrm{vec}(A^\top)$.

For the first term in the objective function (4.6), it is possible to write that:

$$\left\langle UU^\top, L \right\rangle = \mathrm{trace}(UU^\top L) = \mathrm{trace}(U^\top L U) \tag{4.9}$$

Therefore

$$\nabla \left\langle UU^\top, L \right\rangle = LU + L^\top U = 2LU \tag{4.10}$$

because L is symmetric.

Consider the second term of the objective function:

$$\mathrm{vec}\left(\frac{\|UU^\top\|_1}{\partial U}\right)^\top = \mathrm{vec}(\mathrm{sign}(UU^\top))^\top \frac{\partial UU^\top}{\partial U} \tag{4.11}$$

where

$$\frac{\partial UU^\top}{\partial U} = (I_{N^2} + T_{N^2 \times N^2})(U \otimes I_N). \tag{4.12}$$

Describe the column vector M as

$$M = \left(\frac{\partial UU^\top}{\partial U}\right)^\top \mathrm{vec}(\mathrm{sign}(UU^\top)), \tag{4.13}$$

hence, the Euclidean derivative of the objective function (4.6) is:

$$\nabla f(U) = 2LU + \beta\,\mathrm{ivec}(M). \tag{4.14}$$

We believe that $\mathrm{ivec}(M)$ could be computed in a easier and even more efficient way, especially in those cases where the involved matrices are large.

Knowing that

$$\|UU^\top\|_1 := \sum_i \sum_j |(UU^\top)_{ij}| \tag{4.15}$$

and that

$$\left(UU^\top\right) = \sum_k U_{ik}(U^\top)_{kj} = \sum_k U_{ik}U_{jk}, \tag{4.16}$$

we are able to write

$$\left[\frac{\partial \|UU^\top\|_1}{\partial U}\right]_{ab} = \frac{\partial}{\partial U_{ab}} \sum_i \sum_j \left| \sum_k U_{ik}U_{jk} \right|. \tag{4.17}$$

We can now distinguish three cases:

$$\begin{cases} i=a,\ k=b & \text{or} \\ j=a,\ k=b & \text{or} \\ i=j=a,\ k=b \end{cases}$$

Hence

$$\begin{aligned} \left[\frac{\partial \|UU^\top\|_1}{\partial U}\right]_{ab} &= \frac{\partial}{\partial U_{ab}}\left[\sum_j |U_{ab}|\,|U_{jb}| + \sum_i |U_{ib}|\,|U_{ab}|\right] \\ &= 2\sum_i |U_{ib}|\,\mathrm{sign}\left[\sum_k |U_{ik}|\,|U_{ak}|\right]. \end{aligned} \tag{4.18}$$

So whatever way you choose to calculate ivec(M), at the representative U of a Grassmann point U, the Riemannian gradient can be simply computed as:

$$\mathrm{grad}_U f = (I - UU^\top)\nabla f(U). \tag{4.19}$$

At this stage, it is possible to use any suitable optimization algorithm on the Grassmann manifold to solve the optimization problem (4.8) to get a solution U.

Given the data matrix $X = [x_1, \dots, x_N]$, the number of the latent dimension p and the trade-off parameter β, GSC algorithm consists of the steps described in Algorithm 5.

4.3 Clustering methods

We will now briefly summarize the basic concepts about *k-means++* and *Ncut*, two of the major clustering methods, leaving to the reader the opportunity to decide between the two of them the one he prefers. We used an *NNut* method as it reaches better performances.

4.3.1 k-means++

The *k-means++* algorithm uses an heuristic to find centroid seeds for the *k-means* clustering. According to Arthur and Vassilvitskii [1], *k-means++* improves the running time of Lloyd's algorithm (*k-means*) and the quality of final solution. Its goal is to minimize the average squared distance between points in the same cluster. The *k-means++* algorithm chooses seeds as shown in the Algorithm 6, assuming the number of cluster is k. The authors of [1] demonstrate that *k-means++* consistently outperforms *k-means*, both by achieving a lower potential value and also by completing faster.

Algorithm 5 Grassmann Manifold Optimization Assisted Spectral Clustering (GSC)

```
1: Construct a $N \times N$ affinity matrix $W$ where each element $w_{ij}$ measures the similarity between $x_i$ and $x_j$.
2: Defining $k$ the number of clusters, compute the initial latent representation $U^{(0)}$ of size $N \times k$ by taking the first $k$ eigenvectors corresponding to the largest $k$ eigenvalues of the matrix $W$.
3: Compute the Laplacian normalized matrix $L$.
4: With the initial $U^{(0)}$ call any appropriate optimization algorithm on the Grassmann manifold to minimize the following function:
       $$f(U) = \left\langle UU^\top, L \right\rangle + \beta \|UU^\top\|_1.$$
5: With the sparse latent representation $U$ in output, form the new affinity matrix $\hat{W} = UU^\top$
6: Using the $\hat{W}$ matrix, compute the pair-wise Euclidean distance
       $$\Delta_{ij} = \sqrt{P_{ij}},$$
   where, upon defining $\mathbb{1}$ as the $1 \times k$ all-one vector, the matrix $P$ is computed as follows:
       $$\begin{cases} H := \hat{W}^\top \hat{W}, \text{ (size } k \times k) \\ Q := \operatorname{diag}(H), \text{ (size } k \times 1) \\ P := Q\mathbb{1} + \mathbb{1}^\top Q^\top - 2H, \text{ (size } k \times k) \\ \text{Each negative entry of the matrix } P \text{ is set to zero.} \end{cases}$$
7: Take the new pair-wise data affinity matrix $W^*$ as $W^*_{ij} := \exp(-\Delta_{ij}/\sigma)$, where $\sigma = 0.1$, as input for the *Normalized Cut* (NCut) method (or any other clustering algorithms) to separate the data into clusters.
```

Algorithm 6 k-means++

```
1: Let $X \subset \mathbb{R}^{d\times n}$ be a set of $n$ data points. Select an observation uniformly at random
   from the dataset $X$: the chosen point is the first center and is denoted $c_1$.
2: Compute distances from each data point to $c_1$. Denote the distance between $c_j$ and
   the data point $m$ as $D(x_m, c_j)$.
3: Select the next center $c_2$ at random from $X$ with probability
       $$\frac{D^2(x_m,c_1)}{\sum_{j=1}^{n} D^2(x_j,c_1)}$$
4: To choose center $j$:
     1. Compute the distances from each data point to each center, and assign each
        data point to its closest center.
     2. For $m = 1,\dots,n$ and $p = 1,\dots,j-1$ select center $j$ at random from $X$ with
        probability
            $$\frac{D^2(x_m,c_p)}{\sum_{(h:x_h\in C_p)} D^2(x_h,c_p)}$$
        where $C_p$ is the set of all the data points closest to center $c_p$ and $x_m$ belongs to
        $C_p$.
5: Repeat the previous step until $k$ centers are chosen.
```

4.3.2 Normalized Cut (NCut)

The set of points in an arbitrary feature space is represented as a weighted undirected graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, where the nodes of the graph are the points in the characteristic space, and an edge is built between every pair of nodes. The weight on each edge $w(i, j)$ is a function of the similarity between nodes i and j. In grouping, the set of vertices is partitioned into disjoint sets $\mathscr{V}_1, \mathscr{V}_2, \dots, \mathscr{V}_m$, where the similarity among the vertices in a set $\mathscr{V}_i$ is high and across different sets $\mathscr{V}_i, \mathscr{V}_j$ is low.

A graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ can be partitioned into two disjoint sets $\mathscr{A}, \mathscr{B}$ with $\mathscr{A} \cup \mathscr{B} = \mathscr{V}$ and $\mathscr{A} \cap \mathscr{B} = \emptyset$, by simply removing edges connecting the two parts. The degree of dissimilarity between these two pieces can be computed as total weight of the edges that have been removed. In a graph-theoretic language, it is called the *cut*:

$$\mathrm{cut}(\mathscr{A}, \mathscr{B}) := \sum_{u\in\mathscr{A}, v\in\mathscr{B}} w(u, v).$$

The optimal bipartitioning of a graph is the one that minimizes this cut value. Even though there are a lot of such partitions, finding the minimum cut of a graph is a well-studied problem and there exist efficient algorithms for solving it.

The *NCut* method [14] measures both the total dissimilarity between the different

Algorithm 7 Normalized Cut (NCut)

```
1: Given an image or image sequence, set up a weighted graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ and set the
   weight on the edge connecting two nodes to be a measure of the similarity between
   the two nodes.
2: Summarize the information into matrices $W$ and $D$.
3: For eigenvectors with the smallest eigenvalues, solve
                    $(D - W)x = \lambda Dx$
   otherwise – that is the same
                    $D^{-1/2}(D - W)D^{-1/2}z = \lambda z$
   where $z = D^{-1/2}x$.
4: Use the eigenvector with the second smallest eigenvalue to bipartition the graph by
   finding the splitting point such that the NCut value is minimized.
5: Decide if the current partition should be subdivided by checking the stability of the
   cut, and make sure that the NCut value is below the pre-specified value.
6: Recursively re-partition the segmented parts if necessary.
```

groups as well as the total similarity within the groups: it computes the *cut* cost as a fraction of the total edge connections to all the nodes in the graph, instead of looking at the value of total edge weight connecting the two partitions:

$$\text{NCut}(\mathscr{A}, \mathscr{B}) := \frac{\text{cut}(\mathscr{A}, \mathscr{B})}{\text{assoc}(\mathscr{A}, \mathscr{V})} + \frac{\text{cut}(\mathscr{A}, \mathscr{B})}{\text{assoc}(\mathscr{B}, \mathscr{V})}$$

where $\text{assoc}(\mathscr{A}, \mathscr{V}) := \sum_{u \in \mathscr{A}, t \in \mathscr{V}} w(u,t)$ is the total connection from nodes in $\mathscr{A}$ to all nodes in the graph and $\text{assoc}(\mathscr{A}, \mathscr{V})$ is similarly defined. With this definition of the disassociation between the groups, the cut that partitions out small isolated points will no longer have small NCut value, since the cut value will almost certainly be a large percentage of the total connection from that small set to all other nodes.

Coming to the point, the *NCut* algorithm consists of the steps specified in the Algorithm 7. The authors of the [14] algorithm make it clear that they have developed a grouping algorithm based on the view that perceptual grouping should be a process that aspires to extract global impressions of a scene, and that provides a hierarchical description of it. By treating the grouping problem as a graph partitioning problem, they proposed a method which is an unbiased measure of disassociation between subgroups of a graph, and it has the nice property that minimizing normalized cut leads directly to maximizing the normalized association, which is – again – an unbiased measure for total association within the sub-groups.

Unlike *k-means* which is ideal for discovering globular clusters, where all members of each cluster are in close proximity to each other (in the Euclidean sense), *NCut* doesn't cluster data points directly in their native data space but instead forms a similarity matrix where the (i, j)-th entry is some similarity distance it defines between the i-th and j-th data points in the dataset. So, in a certain sense, the *NCut* algorithm is more general (and powerful) because whenever *k-means* is appropriate for use then so too is *NCut* (just use a simple Euclidean distance as the similarity measure). The converse is not true though.

Chapter 5

Experimental results

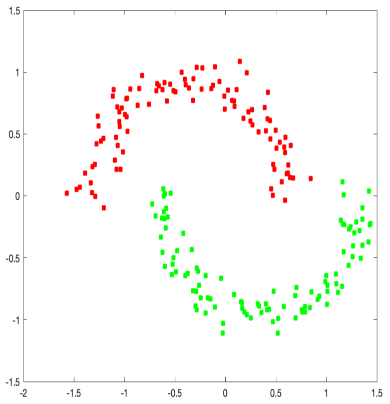
The present chapter summarizes results of numerical experiments performed on two categories of datasets, namely, synthetic 2-dimensional datasets used for testing purposes, and real-world datasets used to validate and compare the performances of the discussed gradient-based learning algorithms.

5.1 Clustering results on synthetic datasets

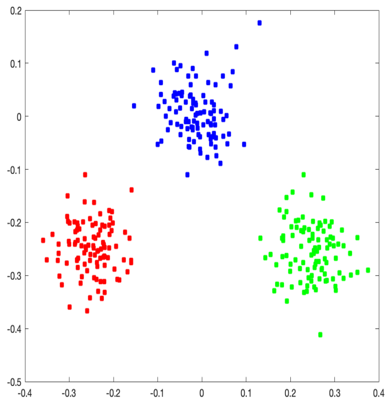
The synthetic datasets used in this section are drawn from [15], namely:

- **Two-moon data**: The data are randomly generated from two sine-shape curves with the noise percentage set to 0.09, and each cluster – in this specific case – contains 100 samples.
- **Three-Gaussian data**: Each cluster follows a Gaussian distribution with a variance of 0.05. Again, in our experiment each cluster has 100 samples.
- **Three-ring data**: The data are distributed on circles, with the noise percentage set to 0.15. There are respectively 100, 100 and 150 samples in each cluster.
- **Two disjoint quadratic para-curves data**: The data are spread throughout two disjoint parabolic-shape curves without overlapping, altered with Gaussian noise of 0 mean and variance 0.05. In our experiment each cluster contains 200 samples.

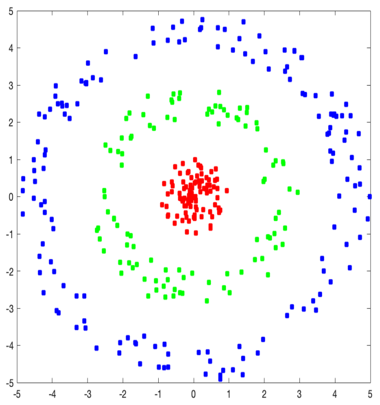
The datasets used in this experiment are shown in the Figure 5.1, with the clusters colored. The numerical experiments were performed on a PC with a dual-core Intel Core i5 processor, clock frequency 2.7GHz and 8GB RAMs by MATLAB R2017b scripts.



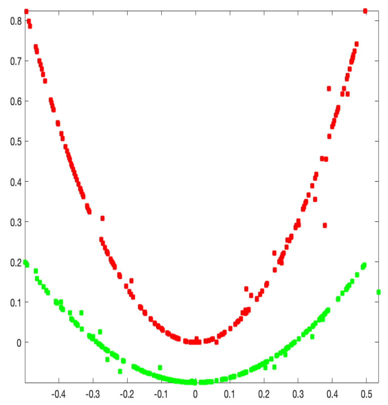
(a) *Two-Moon dataset.*



(b) *Three-Gaussian dataset.*



(c) *Three-Ring dataset*



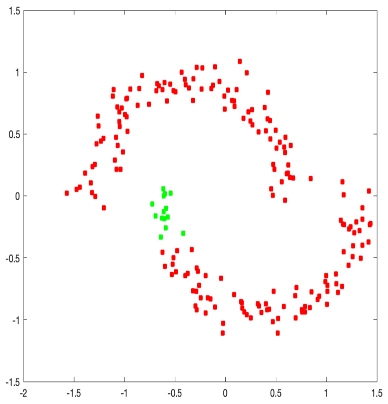
(d) *Two disjoint quadratic para-curves dataset*

Figure 5.1: Synthetic datasets used to test clustering algorithms.

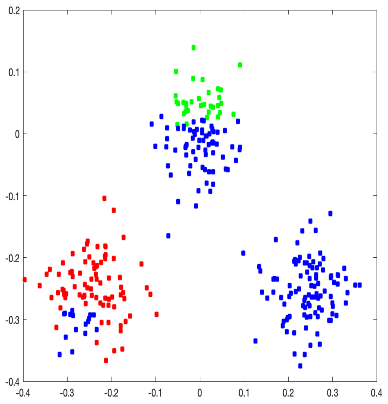
Before discussing and comparing the efficiency of the various algorithms on each dataset, the *NCut* limitations deserve a closer look: its accuracy is not high in any of the dataset presented, as the Figure 5.2 shows.

Once the importance of these optimization algorithms has been clarified, we proceed in comparing them. Each of the following tables refers to one of the methods listed above, and it shows both the number of iterations and the running time that the considered algorithm needs to get the 100% clustering accuracy for each dataset.

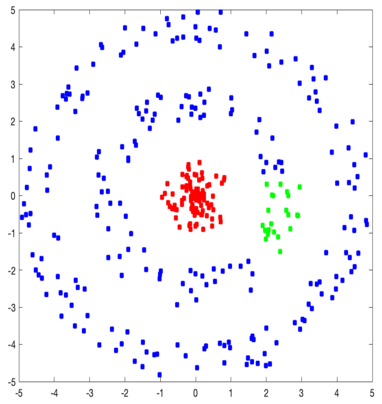
First, we compare TrustRegions ManOpt tool, using both matrix M of the paper [15] (4.13), whose performances are summarized in the Table 5.1, and then our improvement (4.18), whose performances are summarized in the Table 5.2.



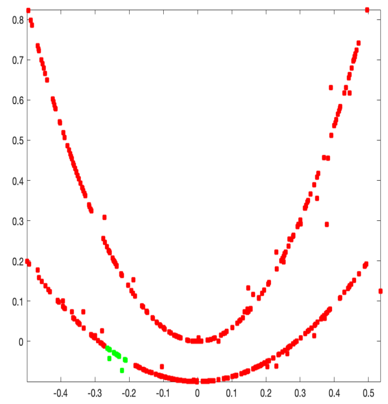
(a) *NCut with Two-moon dataset.*



(b) *NCut with Three-Gaussian dataset.*



(c) *NCut with Three-ring dataset.*



(d) *NCut with Two disjoint para-curves dataset.*

Figure 5.2: Clustering results obtained by the *NCut* algorithm without learning any affinity matrix.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	17	60.75
Three-Gaussian	8	162.87
Three-Ring	11	304.10
Two Disjoint Para-Curves	33	1047.79

Table 5.1: Comparison of the performance indices of the TrustRegions (ManOpt tool) using (4.13).

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	12	4.59
Three-Gaussian	5	2.59
Three-Ring	10	3.42
Two Disjoint Para-Curves	11	4.39

Table 5.2: Comparison of the performance indices of the TrustRegions with our improvement in computation of M (4.18).

As we can see, our improvement in the way to compute M (4.18) makes the algorithm much faster.

Notice that the number of iterations is not so meaningful, as the running time of the algorithm is. The more complex the toy dataset is, the more time the algorithm requires to converge perfectly, regardless of whether the number of iterations is slightly or significantly higher than that used in other datasets. We shall now move on to the gradient-based learning methods. The following results were obtained by choosing, as calculation method for the matrix M, the expression (4.18).

The performances of the ‘Gradient descent’ learning algorithm on the four synthetic datasets are summarized in the Table 5.3. In this table, as well as in the following ones, the # iterations refers to a pre-fixed number of learning cycles which guarantees the complete convergence. It is clear that the convergence of this algorithm is very slow.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	200,000	567.05
Three-Gaussian	500,000	1,463.19
Three-Ring	750,000	9,617.48
Two Disjoint Para-Curves	500,000	5,277.69

Table 5.3: Comparison of the performance indices of the GD-based learning algorithm on four synthetic datasets.

The performances of the ‘Momentum’ learning algorithm on the four toy datasets are summarized in the Table 5.4. ‘Momentum’ is much faster than ‘GD’, but it is still not moving fast.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	3,500	12.42
Three-Gaussian	1,500	18.28
Three-Ring	250,000	3,023.19
Two Disjoint Para-Curves	300,000	3,675.03

Table 5.4: Comparison of the performance indices of the Momentum-based learning algorithm on four synthetic datasets.

The performances of the 'Nesterov accelerated gradient' learning algorithm on the four toy datasets are summarized in the Table 5.5. As expected, 'Nesterov accelerated gradient' is similar to 'Momentum' in terms of performance, although it appears to be slightly faster as the complexity of the input data increases.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	6,500	21.86
Three-Gaussian	1,000	13.04
Three-Ring	10,000	156.30
Two Disjoint Para-Curves	10,000	173.70

Table 5.5: Comparison of the performance indices of the NAG-based learning algorithm on four synthetic datasets.

The performances of the 'Adaptive gradient' learning algorithm on the four toy datasets are summarized in the Table 5.6. Adaptive methods turn out to be much faster, as it is possible to notice by observing the performance of the 'AdaGrad', which is however the slowest of them.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	2,000	19.58
Three-Gaussian	200	8.92
Three-Ring	300	15.34
Two Disjoint Para-Curves	5,000	77.59

Table 5.6: Comparison of the performance indices of the AdaGrad-based learning algorithm on four synthetic datasets.

The performances of the 'AdaDelta' learning algorithm on the four toy datasets are summarized in the Table 5.7. 'AdaDelta' still improves 'AdaGrad' performances and turns out to be faster.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	1,000	5.91
Three-Gaussian	100	2.41
Three-Ring	400	7.08
Two Disjoint Para-Curves	3,500	32.83

Table 5.7: Comparison of the performance indices of the Adadelta-based learning algorithm on four synthetic datasets.

The performances of the 'AdaM' learning algorithm on the four toy datasets are summarized in the Table 5.8. 'AdaM'appears to be the fastest method, whatever the complexity of the input data is.

Dataset	**# iterations**	**Running time** (sec)
Two-Moon	350	3.74
Three-Gaussian	100	4.34
Three-Ring	100	6.06
Two Disjoint Para-Curves	500	14.55

Table 5.8: Comparison of the performance indices of the AdaM-based learning algorithm on four synthetic datasets.

Apparently, the adaptive methods are much faster than the non-adaptive gradient-based methods. In particular, the *AdaM* learning algorithm seems to be the one which converges the fastest, whereas the *GD* – as well as *Momentum* and *NAG* in the case of the Three-ring dataset – appear to be inappropriate because of their slowness.

The Figure 5.3 shows a comparison of learning curves of six learning algorithms on a Two-moon dataset. The number of iterations is limited to 350, which corresponds to the number of iterations needed by the *AdaM* algorithm to converge to a sparse matrix U that guarantees 100% clustering accuracy.

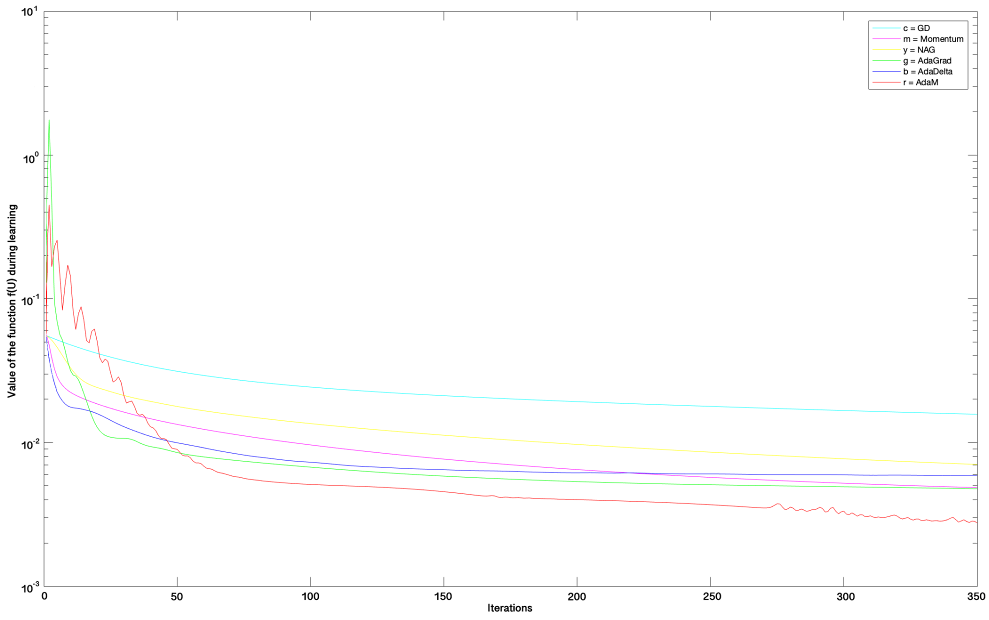


Figure 5.3: Complete view of the learning curves.

The clustering accuracy of the classical *NCut* algorithm on two-moon, three-Gaussian, three-ring datasets is 56.50%, 60.10%, and 86.00%, respectively. We compare the influence of β on the performance of *GSC* learnt by different adaptive methods: *AdaGrad*, *AdaDelta* and *AdaM*. We use for each of the presented optimization method the number of iteration they require to get the perfect accuracy with $\beta = 0.00001$. Tables 5.9, 5.10, 5.11 and 5.12 show the clustering performance versus parameter β on the toy datasets. Each of the considered case states that *GSC* learnt by *AdaM* performance is much more robust than others against the parameter β.

β	**AdaGrad**	**AdaDelta**	**AdaM**
$\beta = 0.00001$	**100%**	**100%**	**100%**
$\beta = 0.00005$	92.75%	**96.50%**	**96.50%**
$\beta = 0.0001$	75.75%	**96.50%**	86.41%
$\beta = 0.0005$	62.41%	66.71%	**78.50%**
$\beta = 0.001$	53.50%	54.25%	**61.41%**
$\beta = 0.005$	53.50%	52.75%	**57.25%**
$\beta = 0.01$	48.41%	49.71%	**51.73%**

Table 5.9: Clustering accuracy of GSC - learnt, respectively, by AdaGrad, AdaDelta and AdaM - against different β on Two-Moon data set.

β	**AdaGrad**	**AdaDelta**	**AdaM**
$\beta = 0.00001$	**100%**	**100%**	**100%**
$\beta = 0.00005$	**100%**	**100%**	**100%**
$\beta = 0.0001$	**100%**	99.75%	**100%**
$\beta = 0.0005$	84.33%	89.50%	**95.70%**
$\beta = 0.001$	60.33%	74.41%	**89.71%**
$\beta = 0.005$	47.25%	74.41%	**88.50%**
$\beta = 0.01$	42.75%	61.71%	**84.33%**

Table 5.10: Clustering accuracy of GSC - learnt, respectively, by AdaGrad, AdaDelta and AdaM - against different β on Three-Gaussian data set.

β	**AdaGrad**	**AdaDelta**	**AdaM**
$\beta = 0.00001$	**100%**	**100%**	**100%**
$\beta = 0.00005$	**100%**	**100%**	**100%**
$\beta = 0.0001$	**100%**	**100%**	**100%**
$\beta = 0.0005$	79.71%	77.14%	**88.50%**
$\beta = 0.001$	69.71%	64.41%	**88.50%**
$\beta = 0.005$	58.43%	55.33%	**83.41%**
$\beta = 0.01$	58.43%	53.65%	**83.41%**

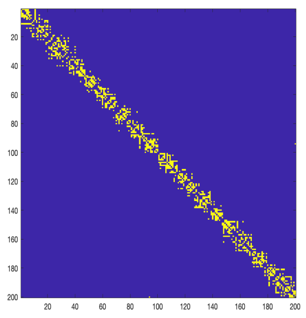
Table 5.11: Clustering accuracy of GSC - learnt, respectively, by AdaGrad, AdaDelta and AdaM - against different β on Three-Ring data set.

For the two disjoint para-curve dataset, the clustering accuracy of *NCut* method is 54.50%. The best result for GSC method is 100% when $\beta = 0.000001$.

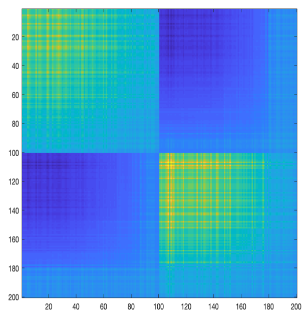
β	**AdaGrad**	**AdaDelta**	**AdaM**
$\beta = 0.000001$	**100%**	**100%**	**100%**
$\beta = 0.000005$	**100%**	**100%**	**100%**
$\beta = 0.00001$	95.75%	96.25%	**97.75%**
$\beta = 0.00005$	65.41%	68.71%	**79.50%**
$\beta = 0.0001$	56.75%	56.75%	**71.40%**
$\beta = 0.0005$	54.50%	55.33%	**61.71%**
$\beta = 0.001$	31.43%	34.66%	**46.75%**

Table 5.12: Clustering accuracy of GSC - learnt, respectively, by AdaGrad, AdaDelta and AdaM - against different β on Two Disjoint Para-Curves data set.

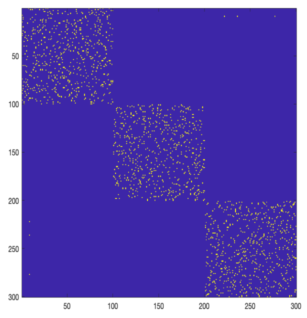
In order to examine the underlying low-dimensional structure within data, we supply here a visual comparison of affinity matrix of all the synthetic datasets used. Figures 5.4 and 5.5 show the affinity matrix W as computed by the *NCut* algorithm and by the *GSC* clustering algorithm (learnt by an *AdaM* algorithm), respectively. Note that the affinity matrix obtained using *AdaM* is the same one we could get employing each one of the other optimization methods. The affinity matrix obtained by *GSC* effectively reveals the cluster structure of data.



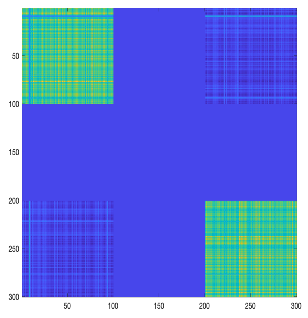
(a) *NCut with Two-Moon dataset.*



(b) *GSC with Two-Moon dataset.*



(c) *NCut with Three-Gaussian dataset.*



(d) *GSC with Three-Gaussian dataset.*

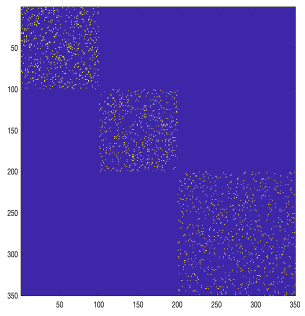
Figure 5.4: Comparison of the affinity matrix W computed on Two-Moon and Three-Gaussian Dataset: (a), (c) were obtained using *NCut*; (b), (d) were obtained using *GSC* learnt by *AdaM*.

From all the results listed above, it is possible to observe that:

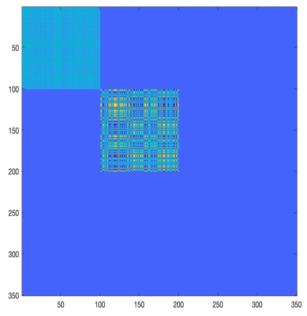
1. Due to our adjustments (equation 3.4), GSC algorithm appears to be clearly faster: looking at tables 5.1, 5.2, it is clear that the running time of the algorithm is considerably lower.

2. The clustering results from GSC are better that from the *NCut* algorithm, for each toy dataset and each optimization algorithm considered.

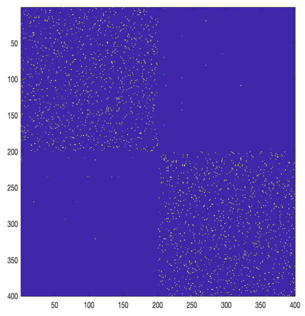
3. *AdaM* seems to be the fastest optimization method: especially for the complicated three-Gaussian, three-ring and two disjoint para-curves datasets it outperforms all the others, and it is also much more robust versus the sparse regularization



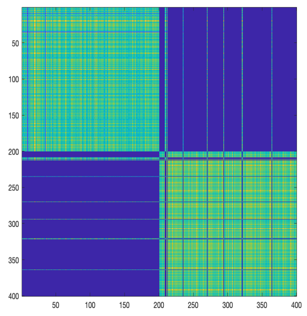
(a) *NCut with Three-Ring dataset.*



(b) *GSC with Three-Ring dataset.*



(c) *NCut with Two-ParaCurves dataset.*



(d) *GSC with Two-ParaCurves dataset.*

Figure 5.5: Comparison of the affinity matrix W computed on Three-Ring and Two Disjoint Para-Curves Dataset: (a), (c) were obtained using *NCut*; (b), (d) were obtained using *GSC* learnt by *AdaM*.

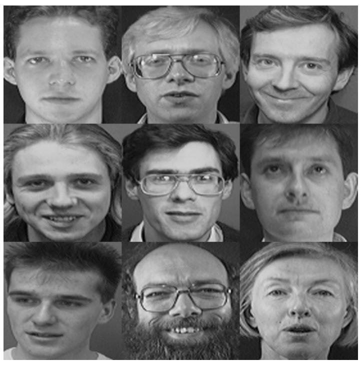
parameter β.

5.2 Clustering results on pictorial datasets

In this section, we perform some experiments on public databases to evaluate the performances of the proposed optimization methods on real-world dataset. All of the experiments are conducted on the following three public available datasets:

1. The YaleB face database (`http://vision.ucsd.edu/~leekc/ExtYaleDatabase/ExtYaleB.html`)
2. The ORL face database (`https://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`)
3. The MNIST database (`http://yann.lecun.com/exdb/mnist/`)

(a) *Examples of Extended Yale B Dataset.*


(b) *Examples of ORL faces Dataset.*

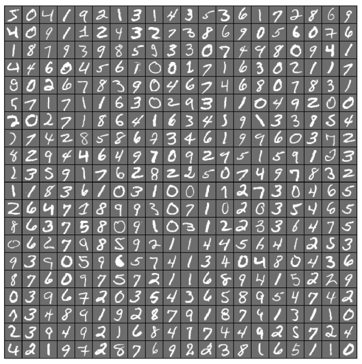

(c) *Examples of MNIST handwritten digits.*

Figure 5.6: Examples of the Extended YaleB, ORL Faces and MNIST datasets, respectively.

The *Extended Yale B Face Database* consists of 192 $\times$ 168 pixel cropped face images of $n = 38$ individuals, where there are $N_i = 64$ frontal face images for each subject acquired under various lighting conditions. Some sample face images are shown in Figure 5.6(a). To reduce the computational cost and the memory requirements of all algorithms, we downsample the images to 32 $\times$ 32 pixels and treat each 1,032-dimensional vectorized image as a data point, hence, $D = 1,032$.

The *ORL Database* is composed of 400 images of size 112 $\times$ 92, and some samples are shown in Figure 5.6(b). There are 10 different images of 40 distinct subjects. For some of the subjects, the images were taken at different times, varying lighting slightly, facial expressions (open/closed eyes, smiling/non-smiling) and facial details (glasses/no-glasses). All the images are taken against a dark homogeneous background and the subjects are in up-right, frontal position (with tolerance for some side movement). All the data is collected in a matrix of shape 10304(pixels) $\times$ 400(faces). To avoid large values, the data matrix is divided by 100.

We follow the settings in [5] to construct the affinity matrix by l_1-graph and apply GSC learnt by ManOpt-Trustregions and AdaM on the constructed affinity matrix for these two face data sets. Six subsets are constructed which consist of all the images of the randomly selected subjects with the number of clusters, i.e., K ranging from 5 to 18. We set the same β = 0.00001, and n = 1500 the number of iterations for the AdaM algorithm.

Method	**Trustregions**	**AdaM**
$K = 5$	96.50%	**97.15%**
$K = 8$	**91.65%**	**91.65%**
$K = 10$	85.24%	**86.33%**
$K = 12$	81.31%	**81.71%**
$K = 15$	**77.82%**	76.91%
$K = 18$	**74.61%**	74.51%

Table 5.13: Clustering results in terms of accuracy of GSC learnt by Trustregions and by AdaM on YaleB dataset.

Method	**Trustregions**	**AdaM**
$K = 5$	**97.60%**	**97.60%**
$K = 8$	93.50%	**94.25%**
$K = 10$	**82.77%**	82.67%
$K = 12$	81.80%	**81.95%**
$K = 15$	**79.67%**	**79.67%**
$K = 18$	**78.86%**	77.95%

Table 5.14: Clustering results in terms of accuracy of GSC learnt by Trustregions and by AdaM on ORL dataset.

The result are summarized in Tables 5.13 and 5.14. They show how close is the accuracy reached by GSC learnt both by Trustregions and AdaM, although AdaM seems to reach a better accuracy with relatively low K whereas Trustregions is better with higher K.

The subset of handwritten digits images in Figure 5.6(c) is selected from *MNIST* database, which contains 60,000 training digital images and 10,000 testing digital images, with 600 images of each digit. All images are normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. We compared the performance of GSC learnt by ManOpt-TrustRegions and adaptive methods. We set the same β = 0.00001, whereas the number of clusters K is set 5, 8 and 10. Each of the K

datasets was build-up so as to contain a total of 400 images randomly selected to belong to the same cluster.

Method	**Trustregions**	**AdaGrad**	**AdaDelta**	**AdaM**
$K = 5$	96.57%	93.71%	94.15%	**97.50%**
$K = 8$	89.41%	85.69%	87.61%	**89.61%**
$K = 10$	**79.24%**	73.21%	75.86%	**79.24%**

Table 5.15: Clustering results in terms of accuracy of GSC learnt by Trustregions and by adaptive-methods on MNIST dataset.

As Table 5.15 demonstrates, GSC algorithm learnt by AdaM outperforms GSC algorithm learnt by all the other methods, even the TrustRegions, showing that AdaM is the best choice, not only with toy datasets but also with pictorial ones.

Chapter 6

Conclusion

This thesis studied and extended a number of classical and modern gradient-based learning methods to a general smooth manifold. After a quick overview of the spectral clustering's world, we also examined the GSC model which adopts Grassmann manifold optimization strategy to optimize the sparse spectral clustering objective in a straightforward way, and found out a way to make it converge faster. Extensive experiments conducted on both toy datasets and several real-world databases demonstrated the effectiveness of adaptive methods, in particular of AdaM, which seems to represent a valid and efficient alternative – in terms of accuracy and fast convergence – to ManOpt tools such as Trustregions.

Acknowledgments

The present research work was completed when I was taking an internship at the Tokyo University of Agriculture and Technology (TUAT, Koganei campus) thanks to a scholarship of the Università Politecnica delle Marche (CampusWorld Program2019) during March-May 2019.
I wish to gratefully thank Prof. Toshihisa Tanaka who made this internship possible.
I wish to thank Prof. Junbin Gao (The University of Sydney) for sharing part of the computer codes to implement the GSC algorithm, as well as Dr. Ehsan Elhamifar (University of California at Berkeley) and René Vidal (The Johns Hopkins University) for sharing part of the computer codes to implement sparse clustering on pictorial data.
I also wish to thank Prof. Simone Fiori, the supervisor of this thesis, for granting me the opportunity to live an unforgettable experience, guiding me throughout the whole work with his expert advices.
Last but not least, I would really like to thank my family, for having supported me in times of need and throughout the university career, and my sister who truly is a source of inspiration for me and the person who I undoubtedly admire the most, as well as all my closest friends, who also encouraged me until the end, playing a fundamental role and often serving as reference figures.

Bibliography

[1] D. ARTHUR AND S. VASSILVITSKII, *K-means++: The advantages of careful seeding*, in SODA 2007: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2007, pp. 1027–1035.

[2] S. BONNABEL, *Stochastic gradient descent on Riemannian manifolds*, IEEE Transactions on Automatic Control, 58 (2013), pp. 2217–2229.

[3] A. BOTEV, G. LEVER, AND D. BARBER, *Nesterov's accelerated gradient and momentum as approximations to regularised update descent*, in 2017 International Joint Conference on Neural Networks (IJCNN), May 2017, pp. 1899–1903.

[4] J. DUCHI, E. HAZAN, AND Y. SINGER, *Adaptive subgradient methods for online learning and stochastic optimization*, Journal of Machine Learning Research, 12 (2011), pp. 2121–2159.

[5] E. ELHAMIFAR AND R. VIDAL, *Sparse subspace clustering: Algorithm, theory, and applications*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 35 (2013), pp. 2765–2781.

[6] S. FIORI, *Learning by natural gradient on noncompact matrix-type pseudo-Riemannian manifolds*, IEEE Transactions on Neural Networks, 21 (2010), pp. 841–852.

[7] S. FIORI, T. KANEKO, AND T. TANAKA, *Tangent-bundle maps on the Grassmann manifold: Application to empirical arithmetic averaging*, IEEE Transactions on Signal Processing, 63 (2015), pp. 155–168.

[8] D.P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, in 2014 International Conference on Learning Representations (ICLR), 2014.

[9] C. LU, S. YAN, AND Z. LIN, *Convex sparse spectral clustering: Single-view to multi-view*, IEEE Transactions on Image Processing, 25 (2016), pp. 2833–2843.

[10] Y.E. NESTEROV, *A method for solving the convex programming problem with convergence rate* $o(1/k^2)$, Dokl. Akad. Nauk SSSR, 269 (1983), pp. 543–547.

[11] A.Y. NG, M.I. JORDAN, AND Y. WEISS, *On spectral clustering: Analysis and an algorithm*, in Advances in Neural Information Processing Systems 14, T.G. Dietterich, S. Becker, and Z. Ghahramani, eds., MIT Press, 2002, pp. 849–856.

[12] N. QIAN, *On the momentum term in gradient descent learning algorithms*, Neural Networks, 12 (1999), pp. 145 – 151.

[13] H. SAMET, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[14] J. SHI AND J. MALIK, *Normalized cuts and image segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 22 (2000), pp. 888–905.

[15] Q. WANG, J. GAO, AND H. LI, *Grassmannian manifold optimization assisted sparse spectral clustering*, in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017, pp. 3145–3153.

[16] M.D. ZEILER, *ADADELTA: an adaptive learning rate method*, CoRR, abs/1212.5701 (2012).