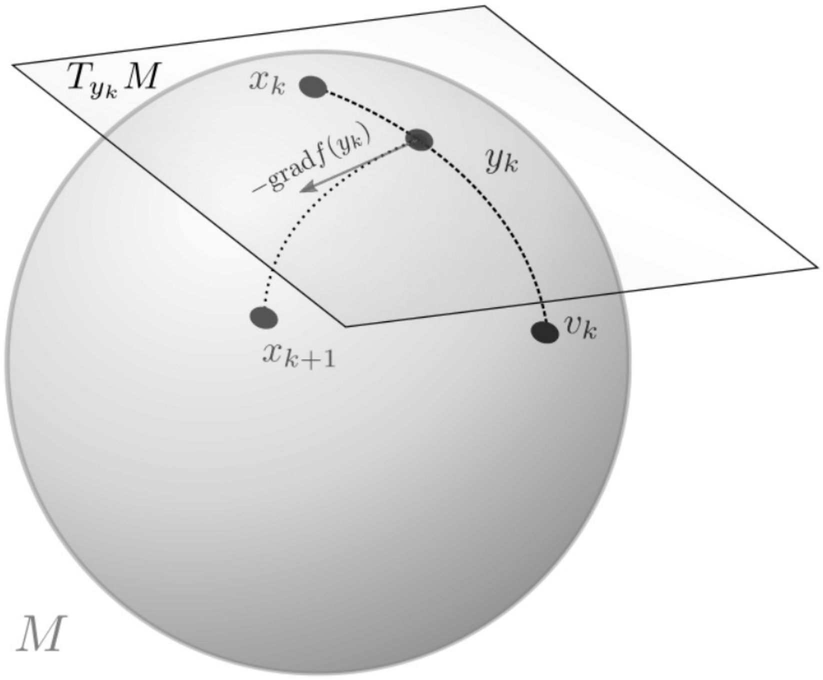
NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Fabian Vakhidi

Pose Estimation with Convolutional Neural Networks

A study of Riemannian optimization with various rotation representations in deep rotation regression using convolutional neural networks.

Master's thesis in Mechanical Engineering
Supervisor: Olav Egeland
June 2022



Fabian Vakhidi

Pose Estimation with Convolutional Neural Networks

A study of Riemannian optimization with various rotation representations in deep rotation regression using convolutional neural networks.

Master's thesis in Mechanical Engineering
Supervisor: Olav Egeland
June 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Acknowledgements

I would like to express my sincere appreciation to my Professor, Olav Egeland, for the counseling during the writing of this master's thesis. I am honored to have been working with such a talented academic, and I wish him the utmost success in all his future endeavours. I would also like to express my deepest gratitude towards my family who have provided continuous support during my tenure as a student.

Abstract

Pose estimation with convolutional neural networks (CNN) falls under the umbrella as deep rotation regression. Deep rotation regression determines a rotation matrix from point cloud measurements, and the solution will depend on the representation that is used for the rotation matrix. In particular, this master's thesis is inspired by the contribution of Chen et al.[1] which studies the gradients of the quaternion, 6D, 9D and 10D representations during the backpropagation stage of a CNN. The simulations conducted in this thesis proves that by employing Riemannian optimization to compute manifold-aware gradients through a goal rotation R_g, consistently improves network performance when using g_M and g_{RPM} on quaternion, 6D, 9D and 10D representations. The simulations shows that the g_{RPM} from 6D, 9D and 10D representations provides the most optimal convergence and neural network learning. The simulations further proves that the homeomorphic rotation representations enjoys the better network performance than their discontinuous counterparts when using Euclidean gradients, g_M and g_{RPM}.

Sammendrag

Positurestimering ved hjelp av convolutional neural networks (CNN) faller under fellesbetegnelsen deep rotation regression. Deep rotation regression bestemmer en rotasjonsmatrise fra punktskyer, hvor løsningen vil sterkt avhenge av representasjonen som brukes for rotasjonsmatrisen. Denne masteroppgaven er inspirert av bidraget fra Chen et al.[1] som studerer gradientene til lærevennlige rotasjonsrepresentasjoner under backpropagation-stadiet til et CNN. Simuleringene utført i denne oppgaven beviser at ved å bruke Riemann-optimalisering for å beregne manifoldbevisste gradienter gjennom en målrotasjon R_g, konsekvent forbedrer nettverksytelsen ved bruken av g_M og g_{RPM} på quaternion, 6D, 9D og 10D representasjonene. Simuleringene viser at g_{RPM} fra 6D, 9D og 10D representasjonene gir mest optimal konvergens. Simuleringene viser også at de homeomorfe rotasjonsrepresentasjonene har bedre nettverksytelse enn deres diskontinuerlige motsetninger når det brukes Euklidiske gradienter, g_M og g_{RPM}.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

The forthcoming of an advanced autonomous world requires the processing of semantic information of the objects in the world around us. The fast development of high precision sensors such as Light Detection and Ranging (LiDAR) has led to point clouds being the primary data format to represent the 3D world [2]. LiDAR captures laser scans of the 3D scene to generate a cloud of spatial information. The cloud (or the data set) is an irregular and unordered composition of 3D-arrays. Despite of these great aspects, LiDAR is constrained to scans of limited view ranges, which creates a dependence of a registration algorithm to gather information of the complete 3D scene. The registration problem involves estimating the rigid-transformation between two point clouds, which is generally known as *pose estimation*. Moreover, LiDAR proves to be ineffective in poor weather conditions, which potentially leaves the point cloud being corrupted with noise and outliers. In order to tackle such obstacles, a registration algorithm must be robust against outliers and precise in its rigid-transformation estimations.

The research community is extensively working towards providing registration algorithms with state-of-the-art performances, as several solutions have been proposed. Recent works from Yang et al.[3] and Zhou et al.[4] with Truncated least squares Estimation And SEmidefinite Relaxation (TEASER) and Fast Global Registration (FGR), respectively, have proven to be quite successful in their domain, and offers high precision and robustness. The resurgence of the deep learning community has offered new proposals by tackling the registration problem through the lens of deep learning-frameworks. Gao et al.[5] was the first to introduce a deep learning based pose estimation (deep rotation regression) method that uses point clouds as inputs in a convolutional neural network (CNN). Their work estimates the rigid-transformation by directly regressing on rotations under supervised learning, where the axis-angle rotation representation is best suited for the learning task. The contribution of Gao et al.[5] has later been extended in Zhou et al.[6], who studies deep learning-friendly rotation representations, where the conclusion is that a rotation representation must be continuous in order to provide correct results when using the whole rotation space. Contributions from Levinson et al.[7] and a case-study from Romain Brégier [8] expands this problem area of learning-friendly representations. Recent work from Chen et al.[1] tackles an under-explored avenue of deep rotation regression by studying the gradients extracted during the backpropagation stage in the neural network, in which a solution of Riemannian optimization is proposed. In this report, all of the aforementioned contributions in deep rotation regression will be extensively studied.

1.1. Notations

$\mathbb{R}$, $\mathbb{N}$ and I are used to denote the set of real numbers, natural numbers and the identity matrix, respectively. The determinant, trace, transpose, inverse, skew-symmetric and Frobenius norm of a matrix A are denoted by $\det(A), \operatorname{tr}(A), A^\top, A^{-1}$, $A^\times$ and $\|A\|_F^2$ respectively. The tangent space of a manifold $\mathcal{M}$ at a point x is denoted using $T_x\mathcal{M}$ and the geodesic distance is given as $d_\mathcal{M}$. The notation $\mathbb{R}^n$ is used to indicate *n-dimensional space*, while Euclidean plane and the Euclidean space are referred to as $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively. $SO(n)$ denotes the Lie group, while the lowercase $\mathfrak{so}(n)$ denotes the Lie algebra. The notations $\exp(\cdot)$ and $\log(\cdot)$ are used to denote the matrix exponential and logarithm, respectively.

Chapter 2.

Background

This chapter serves the theoretical background for the implementations presented later in this thesis. Based on this background information, one should be able to apprehend the theory on Lie groups and its corresponding Lie algebra, Singular Value Decomposition (SVD), QR-Decompisition with Gram-Schmidt, various rotation representations and distance measure on $SO(3)$. Topology along with concepts in differential geometry are also presented in order to understand the theory on Riemannian optimization.

2.1. Lie groups

2.1.1. General Lie groups

A *Lie group* G is a topological group and a smooth manifold such that group multiplication $G \times G \to G(x, y) \mapsto x \cdot y$ and group inversion $G \to Gx \mapsto x^{-1}$ are smooth maps.

2.1.2. Matrix Lie group

The matrix Lie group is a subgroup G of the general linear group $GL(n, \mathbb{R})$, i.e $G \subseteq GL(n, \mathbb{R})$. Then G is a subset of square invertible matrices of size $n \times n$ with real entries on which smooth maps of matrix multiplication and inversion can be safely used without going outside the subset. It is noted that

$$I_n \in G, \quad \forall g \in G, g^{-1} \in G \quad \text{and} \quad \forall a, b \in G, ab \in G$$

A matrix A is said to be square, symmetric and skew-symmetric when $A \in \mathbb{R}^{n \times n}$, $A = A^T$ and $A = -A^T$, respectively [9].

2.1.3. Special orthogonal group $SO(3)$ and $SO(2)$

The special orthogonal group $SO(n)$ is the set of all square real matrices R, which are represented by $n \times n$ rotation matrices. 3D rotations are expressed as

$$SO(3) = \left\{ R \in \mathcal{M}_{3\times 3}(\mathbb{R}) \mid RR^T = I_3 \mid \det(R) = 1 \right\}, \tag{2.1}$$

where I_3 is the identity matrix of $\mathbb{R}^{3\times 3}$. Similarly, the set of 2D special orthogonal rotation matrices are a subgroup of $SO(3)$ and are denoted as $SO(2)$. The corresponding Lie algebra (tangent space) is $\mathfrak{so}(n)$, where n is the same dimension as its Lie group $SO(n)$. The tangent space for $SO(3)$ is noted in [10] as

$$\mathfrak{so}(3) = \left\{ \Omega \in \mathcal{M}_{3\times 3}(\mathbb{R}) \mid \Omega = -\Omega^T \right\}, \tag{2.2}$$

The logarithm is expressed as

$$\log_{\mathfrak{so}(3)} \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{pmatrix} = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{pmatrix}^{\times} = \begin{pmatrix} 0 & -\xi_3 & \xi_2 \\ \xi_3 & 0 & -\xi_1 \\ -\xi_2 & \xi_1 & 0 \end{pmatrix}. \tag{2.3}$$

The logarithm $\log(R) = \theta k^{\times}$ is computed in [11] as

$$\log(R) = \frac{\arcsin(\|w\|)}{\|w\|}\hat{w}, \quad \hat{w} = \frac{1}{2}\left(R - R^{\mathrm{T}}\right). \tag{2.4}$$

The matrix exponential is in [12] given by

$$R = \exp_{SO(3)} u, \quad u = \log(R), \tag{2.5}$$

where $u \in \mathfrak{so}(3)$ is a local parameterization of the rotation matrix R.

Consider the instance where the logarithm is given by $u = \theta k$ where $k \in \mathbb{R}^3$ is a unit vector. Then R is a rotation matrix rotated by an angle of θ about k given as the matrix exponential defined by the Rodrigues' equation

$$R = I + \sin\theta k^{\times} + (1 - \cos\theta) k^{\times} k^{\times}, \tag{2.6}$$

where $k^{\times}$ is the skew-symmetric representation of k.

2.2. Norms

2.2.1. ℓ_p-norm

A normed linear space $(X, \|\cdot\|)$ is in [13] a linear space X equipped with a norm $\|\cdot\|$. Let x, y be points in X. A norm on X is a real-valued function $\|x\| : \mathbb{R}^n$ where $x \in X$ which fulfills the following

1. Positivity:
$$\|x\| \geq 0, \forall x \in \mathbb{R}^n, \tag{2.7}$$

2. Positive definitness:
$$\|x\| = 0 \Leftrightarrow x = 0, \tag{2.8}$$

3. Homogenity:
$$\|\alpha x\| = |\alpha| \|x\|, \forall \alpha \in \mathbb{R}^n, \tag{2.9}$$

4. The triangle inequality:
$$\|x + y\| \leq \|x\| + \|y\|, \forall x, y \in \mathbb{R}^n, \tag{2.10}$$

where the function $d(x, y) = \|x - y\|$ is a metric on its space X, and returns the distance between x, y as a straight line. The ℓ_p-norm is a general set of norms determined by p, and is noted in [14] as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}, \quad p \geq 1, \tag{2.11}$$

which gives the ℓ_2-norm (Euclidean norm) as

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \quad p = 2. \tag{2.12}$$

Norms and distances in $\mathbb{R}^3$ expresses the normed linear space as $(\mathbb{R}^3, \|\cdot\|)$, are commonly given by the Euclidean norm. Let the vector $a = [a_1, a_2, a_3]^{\mathrm{T}} \in \mathbb{R}^3$. The Euclidean norm $\|a\| = \sqrt{a_1^2 + a_2^2 + a_3^2}$. Let $b = [b_1, b_2, b_3]^{\mathrm{T}}$ also be an element in $\mathbb{R}^3$. Then the distance is given by the Euclidean norm as

$$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2}. \tag{2.13}$$

2.2.2. Frobenius norm

The norm of a matrix is called the Frobenius norm. The Frobenius norm for a matrix $A = \{a_{ij}\} \in \mathbb{R}^{m \times n}$ is defined in [14] as

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}. \tag{2.14}$$

The Frobenius norm is often times used as a loss function to penalize the error in deep neural network and machine learning applications, which is often seen to be

$$\sum_{i=1}^{n} \|y_i - Rx_i\|^2 = \|Y - RX\|_F^2, \tag{2.15}$$

where $Y - RX$ is the sum of the square elements in $\| \cdot \|_F^2$.

Angular distance

Let R_1 and R_2 be two rotation matrices with orientations distinct from each other. The angular distance function is based on the axis-angle parameterization Equation 2.6. Consider the incremental rotation (θ_e, k_e) as

$$R_e = R_1^{\mathrm{T}} R_2 = \exp(\theta_e k_e^{\times}). \tag{2.16}$$

The angular distance is given by the smallest angle of rotation between R_1 and R_2. Let $d_a(R_1, R_2)$ denote the angular distance between two rotation matrices. The angular distance is then noted in [15] as

$$d_a\left(R_1, R_2\right) = d_a\left(I, R_1^{\mathrm{T}} R_2\right) = d\left(I, R_e\right) = |\theta_e| \in [0, \pi]. \tag{2.17}$$

The angular distance is given by the norm of the vector form imposed by the rotation logarithm as

$$d_a(I, R_e) = \|\theta k\|, \tag{2.18}$$

while the matrix form is given by the Frobenius norm of the logarithm as

$$d_a(I, R_e) = \frac{1}{\sqrt{2}} \left\|\theta k^{\times}\right\|_F. \tag{2.19}$$

It follows that the angular distance can be given by the Frobenius norm of the logarithm in Equation 2.3 as

$$d_a(I, R_e)^2 = \frac{1}{2} \|\log(R_e)\|_F^2 = \|u\|^2, \tag{2.20}$$

where $u^\times = \log(R_e)$

2.3. Singular value decomposition

The Singular Value Decomposition (SVD) of a rotation matrix $A \in \mathbb{R}^{n\times n}$ is in [16] given by

$$A = U\Sigma V^{\mathrm{T}}, \tag{2.21}$$

where

$$U \in \mathbb{R}^{n\times n}, \quad \Sigma \in \mathbb{R}^{n\times n}, \quad V \in \mathbb{R}^{n\times n}. \tag{2.22}$$

The matrices U and V are orthogonal matrices given by

$$U = (u_1, \ldots, u_n) \text{ and } V = (v_1, \ldots, v_n). \tag{2.23}$$

The matrix Σ is the square diagonal matrix

$$\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{n\times n}, \tag{2.24}$$

with the singular values along the diagonal.

2.4. QR decomposition with Gram-Schmidt

It is noted in [17] that the QR decomposition of a matrix is a decomposition of the matrix into an orthogonal matrix and a triangular matrix. A QR decomposition of a real square matrix A is a decomposition of A as

$$A = QR, \tag{2.25}$$

where Q is an orthogonal matrix and R is an upper triangular matrix. If A is non-singular (determinant not equal to zero) the decomposition is unique. There exists several proposals for solving the QR decomposition. The Gram-Schmidt orthogonalization is one solution.

Consider the Gram-Schmidt procedure, with the vectors to be considered in the process stacked as columns of the matrix A defined as

$$A = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right] \tag{2.26}$$

Then,

$$u_1 = a_1, \quad e_1 = \frac{u_1}{\|u_1\|}, \tag{2.27}$$

$$u_2 = a_2 - (a_2 \cdot e_1)\, e_1, \tag{2.28}$$

$$e_2 = \frac{u_2}{\|u_2\|}, \tag{2.29}$$

$$u_{k+1} = a_{k+1} - (a_{k+1} \cdot e_1)\, e_1 - \cdots - (a_{k+1} \cdot e_k)\, e_k, \tag{2.30}$$

$$e_{k+1} = \frac{u_{k+1}}{\|u_{k+1}\|}. \tag{2.31}$$

Finally, the QR decomposition returns

$$A = [a_1\,|a_2|\cdots|\,a_n] = [e_1\,|e_2|\cdots|\,e_n] \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix} = QR. \tag{2.32}$$

2.5. Rotation representations

An n-dimensional vector in $\mathbb{R}^n$ can be mapped to a rotation matrix $R \in SO(3)$ by a parameterization noted as ϕ, s.t $\phi : \mathbb{R}^n \rightarrow R \in SO(3)$. The following rotation representations introduces various parameterization procedures mapping n-dimensional vectors to rotation matrices. This includes Euler angles, axis-angle, quaternion, 5D, 6D, 9D- and 10D representations. The parameterization procedures are given in Python scripts in appendix B.5.2.

2.5.1. Euler angles

From [18], consider a succession of three rotations (α, β, γ) about the elementary $x-y-z$ axes, respectively. One can then define the parameterization as $(\alpha, \beta, \gamma) \in \mathbb{R}^3 \to R_x(\alpha)R_y(\beta)R_z(\gamma) \in SO(3)$, where

$$\mathrm{R_x}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}, \quad \mathrm{R_y}(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$
$$\mathrm{R_z}(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.33}$$

2.5.2. Axis-angle

Any arbitrary 3D vector can be mapped to the rotation space through the exponential map in Equation 2.6 [8].

2.5.3. Unit quaternion

Note that the n dimensional unit sphere is given as $S^n = \{x \in \mathbb{R}^{n+1} : \|x\| = 1\}$. It is noted in [1] that unit quaternions represent a rotation using a 4D unit vector $q \in \mathcal{S}^3$ double covering the non-Euclidean 3-sphere in which q and -q identify the same rotation. The corresponding manifold mapping is usually chosen to be a normalization step $\pi_q(x) = x/\|x\|$. Its parameterization ϕ_{4D} converts the unit quaternion q into a rotation matrix given as

$$\phi(q) = \begin{pmatrix} 2\left(q_0^2 + q_1^2\right) - 1 & 2\left(q_1 q_2 - q_0 q_3\right) & 2\left(q_1 q_3 + q_0 q_2\right) \\ 2\left(q_1 q_2 + q_0 q_3\right) & 2\left(q_0^2 + q_2^2\right) - 1 & 2\left(q_2 q_3 - q_0 q_1\right) \\ 2\left(q_1 q_3 - q_0 q_2\right) & 2\left(q_2 q_3 + q_0 q_1\right) & 2\left(q_0^2 + q_3^2\right) - 1 \end{pmatrix} \tag{2.34}$$

where $q = (q_0, q_1, q_2, q_3) \in \mathcal{S}^3$ In the reverse direction, the representation mapping

$\psi(R)$ can be expressed as

$$\begin{cases} q_0 = \sqrt{1 + R_{00} + R_{11} + R_{22}}/2 \\ q_1 = (R21 - R_{12})\,/\,(4 * q_0) \\ q_2 = (R_{02} - R_{20})\,/\,(4 * q_0) \\ q_3 = (R_{10} - R_{01})\,/\,(4 * q_0) \end{cases} \tag{2.35}$$

Note that $q = (q_0, q_1, q_2, q_3)$ and $-q = (-q_0, -q_1, -q_2, -q_3)$ as both are well-founded quaternions parameterizing the same rotation matrix R.

2.5.4. 6D representation and Gram-Schmidt orthogonalization

6D rotation representation, lying on Stiefel manifold $\mathcal{V}_2(R^3)$, uses two orthogonal unit 3D vectors $(\hat{c}_1, \hat{c}_2)$ to represent a rotation, which are essentially the first two columns of a rotation matrix. Its manifold mapping π_{6D}, initiated by Zhou et al.[6], is done through a Gram-Schmidt-like orthogonalization. The Gram-Schmidt procedure from [6] is a modification of the original Gram-Schmidt which was introduced in Section 2.4. The modification from Equation 2.31 is that the last column is generalized to be the cross product $\hat{c}_1 = (\hat{c}_1, \hat{c}_2)$, which gives the parameterization ϕ_{6D}. Its inverse representation mapping ψ_{6D} is given by discarding the third column $\hat{c}_3$ from the rotation matrix, denoted as

$$\psi_{\text{6D}}\left(\begin{bmatrix} | & & | \\ \hat{c}_1 & \dots & \hat{c}_n \\ | & & | \end{bmatrix}\right) = \begin{bmatrix} | & & | \\ \hat{c}_1 & \dots & \hat{c}_{n-1} \\ | & & | \end{bmatrix} \tag{2.36}$$

2.5.5. 5D representation

Zhou et al.[6] proved that the 6D representation could actually be compressed into a 5D representation through the use of stereographic projection combined with normalization, while retaining the continuous properties. Figure 2.1 depicts a stereographic projection of a point p on the unit-sphere S_1, a procedure generalized to lower dimensions, but can easily be transferred to higher dimensions [6]. Let p be a point projected to a sphere by a normalization step, which gives the point N_0 at $(0, 1)$. N_0 is then stereographically projected through an intersection with p and onto the plane $y = 0$, which gives p'. The point p' is a stereographic projection of the initial point p. The combination of steps is referred to in [6] as a *normalized projection*, and is mathematically defined as $P : \mathbb{R}^m \to \mathbb{R}^{m-1}$:

$$P(u) = \left[\frac{v_2}{1 - v_1}, \quad \frac{v_3}{1 - v_1}, \quad \dots, \quad \frac{v_m}{1 - v_1}\right]^T, v = u/\|u\| \tag{2.37}$$

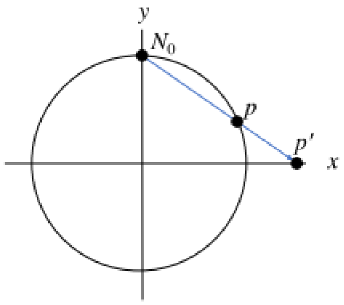


Figure 2.1.: Stereographic in 2D on the unit sphere S_1. Figure is from [6].

A stereographic inverse projection gives the function $Q : \mathbb{R}^{m-1} \to \mathbb{R}^m$ which is noted as

$$Q(u) = \frac{1}{\|u\|}\left[\frac{1}{2}\left(\|u\|^2 - 1\right), \quad u_1, \quad \dots, \quad u_{m-1}\right]^T \tag{2.38}$$

It is noted in [6] that it is possible to make between 1 and $n-2$ projections on an n-dimensional vector while preserving a continuous representation in $SO(n)$. For 3D rotations in $SO(3)$, the 5D representation is a special case of the 6D representation. The 5D representation is obtained by flattening the representation mapping ψ_{6D} to obtain $r \in \mathbb{R}^6$, and then employ normalized projection on the 4 last points in r, which gives $r \in \mathbb{R}^5$. The normalized projected points are then passed through a stereographic inverse projection $Q : \mathbb{R}^5 \to \mathbb{R}^6$, which gives $r \in \mathbb{R}^6$. r is then passed through the aforementioned Gram-Schmidt-like process in Subsection 2.5.4.

2.5.6. 9D representation and SVD orthogonalization

Mapping a 9D representation M to a rotation matrix, Levinson et al.[7] employs SVD orthogonalization as the manifold mapping function π_{9D}. The mapping function π_{9D} first decomposes M into left and right singular vectors $\left\{U, V^\top\right\}$ and singular values Σ, $M = U\Sigma V^\top$. The Σ is then replaced with $\Sigma' = \text{diag}\left(1, 1, \det\left(UV^\top\right)\right)$ and finally, computes $R = U\Sigma' V^\top$ to get the corresponding rotation matrix $R \in \text{SO}(3)$. Note that this representation manifold $\mathcal{M}$ is SO(3), which yields the following rotation mapping as the identity matrix I.

2.5.7. 10D representation

Peretroukhin et al.[19] proposed a 10D representation for rotation matrix. The manifold mapping function π_{10D} maps $\theta \in R^{10}$ to q $\in \mathcal{S}^3$ by computing the eigenvector corresponding to the smallest eigenvalue of $A(\theta)$, expressed as

$\pi_{10D}(x) = \underset{q \in \mathcal{S}^3}{\arg\min}\, q^\top A(x) q$, in which

$$A(\theta) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \\ \theta_2 & \theta_5 & \theta_6 & \theta_7 \\ \theta_3 & \theta_6 & \theta_8 & \theta_9 \\ \theta_4 & \theta_7 & \theta_9 & \theta_{10} \end{bmatrix}. \tag{2.39}$$

Since the representation manifold is $\mathcal{S}^3$, the rotation and representation mapping are the same as unit quaternion Equation 2.34.

2.6. Topology

Topology is the area of mathematics which studies continuity. Objects are considered topologically equivalent if they can be continuously deformed into one another through motions in space such as bending, twisting, stretching, and shrinking while disallowing tearing apart or gluing together parts. The main topics of interest in topology are the properties that remain unchanged by such continuous deformations [20].

2.6.1. Surjectivity and homeomorphism

Let the set $\mathcal{X}$ be a domain and the set $\mathcal{Y}$ be a codomain. A surjective function is in [21] a continuous function f that maps an element $x \in \mathcal{X}$ to every $y \in \mathcal{Y}$. The function is said to be surjective if

$$f : \mathcal{X} \to \mathcal{Y}, \quad \text{if} \tag{2.40}$$

$$\forall y \in \mathcal{Y}, \exists x \in X, \quad f(x) = y \tag{2.41}$$

If (x, y) belongs to the function f, then y is referred to as the *image* of x under f, and x is the *pre-image* of y under f. Using the definition of surjective functions introduced in Equation 2.40, a surjective function is bijective if there exists a continuous inverse f^{-1} which maps elements from the codomain back to the domain, i.e $f^{-1} : \mathcal{Y} \to \mathcal{X}$. A bijective map further leads to the term *homeomorphism*. Two spaces are called topologically equivalent if there exists a homeomorphism between the sets. A homeomorphism preserves the properties between the sets in a one-to-one correspondence. Surjective functions are either one-to-one, one-to-many or many-to-one correspondences. One-to-many/many-to-one correspondences are referred to as non-injective surjective functions, and thus are not homeomorphic. Figure 2.2 shows a bijective and an non-injective surjective correspondence.

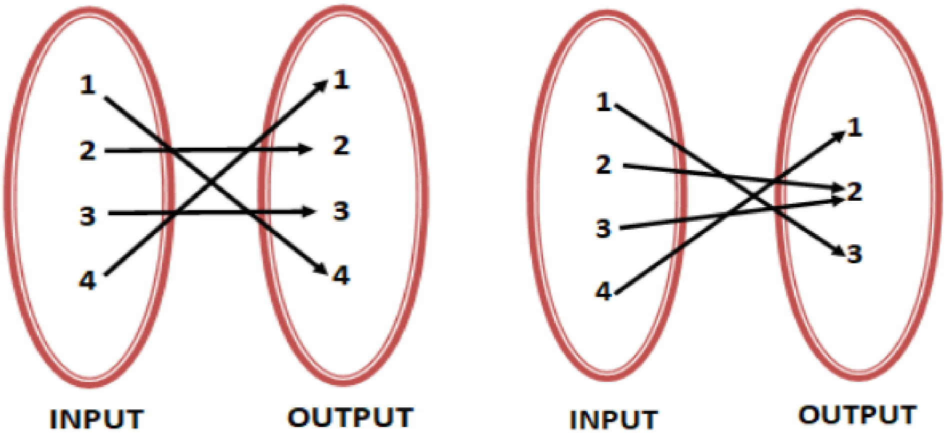


Figure 2.2.: Surjective functions. Figure is from [22].

2.6.2. $SO(n)$ and homeomorphism

Determining homeomorphism between two topological structures requires the introduction of path- and simply-connected manifolds. A topological space $\mathcal{X}$ is called path-connected if for every pair of points $\forall x, y \in \mathcal{X}$ there exists a path γ in $\mathcal{X}$ joining x to y. A topological space is simply-connected if it is path-connected and every path between two points can be continuously transformed into any other such path while preserving the two endpoints in question [23]. $SO(3)$ and $SO(2)$ are path-connected, but not simply-connected. The n-sphere is simply-connected. Thus, the $SO(n)$ manifold is not homeomorphic to any subset of R^n when $n < 4$ [24].

2.7. Differential geometry

2.7.1. Topological- and smooth manifolds

An *n-dimensional manifold* is a topological space $\mathcal{M}$ for which every point $x \in \mathcal{M}$ has a local neighbourhood homeomorphic to Euclidean space $\mathbb{R}^n$ [25]. A topological manifold $\mathcal{M}$ is a non-Euclidean geometric structure. The torus in Figure 2.3 is an example of a topological manifold.

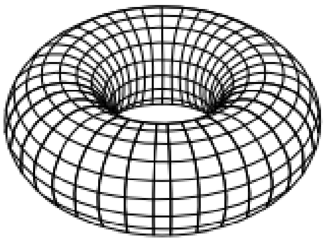


Figure 2.3.: The torus.

Let the torus be a topological manifold $\mathcal{M}$. Each point $x \in \mathcal{M}$ is located in a local neighbourhood, or an open subset $U \subseteq \mathcal{M}$, which is homeomorphic to an open subset of $\mathbb{R}^n$ [26]. The more formal definition of a topological manifold is given in [25] as

1. $\mathcal{M}$ is Hausdorff space, that is, for each distinct point x_n at $\mathcal{M}$, there exists a local neighborhood U_n that separates each point.
2. Each point x_n at $\mathcal{M}$ has a local neighborhood U_n homeomorphic to an open subset $U_\alpha \subseteq \mathbb{R}^n$.
3. $\mathcal{M}$ is *second countable*. The notion of second countable restricts the number of open sets $\mathcal{M}$ can possess.

Let $\mathcal{M}$ be a topological space and $\mathcal{U} \subseteq \mathcal{M}$ an open set. Let $\mathcal{V} \subseteq \mathbb{R}^n$ be open. A homeomorphism $\phi : \mathcal{U} \to \mathcal{V}, \phi(u) = (x_1(u), \dots, x_n(u))$ is called a coordinate system on $\mathcal{U}$, and the functions $x_1, \dots x_n$ the coordinate functions [26]. The pair $(\mathcal{U}, \phi)$ is called a chart on $\mathcal{M}$. The inverse map ϕ^{-1} is a parameterization of $\mathcal{U}$.

An atlas on $\mathcal{M}$ is a collection of charts $\{\mathcal{U}_\alpha, \phi_\alpha\}$ such that $\mathcal{U}_\alpha$ cover $\mathcal{M}$. The homeomorphisms $\phi_\beta \phi_\alpha^{-1} : \phi_\alpha (\mathcal{U}_\alpha \cap \mathcal{U}_\beta) \to \phi_\beta (\mathcal{U}_\alpha \cap \mathcal{U}_\beta)$ are the transition maps or coordinate transformations [26]. A homeomorphism implies that all topological properties are preserved after a transition map

A topological manifold is a *smooth manifold* if all transition maps are $C^\infty(M, x)$ diffeomorphisms, that is, all partial derivatives at point $x \in \mathcal{M}$ exist and are continuous [26].

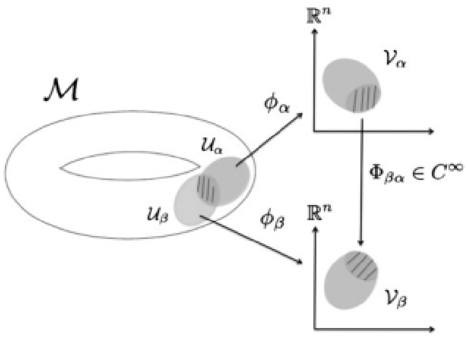


Figure 2.4.: Transition maps.

From [27], a derivation on $C^\infty(M, x)$ is a linear map $\delta : C^\infty(M, x) \to \mathbb{R}^n$, and is denoted by $\mathcal{D}^\infty(M, x)$ as the set of all derivations. $\mathcal{D}^\infty(\mathcal{M}, x)$ is called the tangent space of M at x, which is further denoted as T_xM. Using the introduction of the matrix exponential and logarithm on $SO(3)$ from Subsection 2.1.3, shows that logarithm map to $\mathfrak{so}(3)$ is a chart, while the exponential map is a parameterization. The tangent space of $SO(3)$ at R is expressed as $T_RSO(3)$.

2.7.2. Riemannian manifolds

The intuition of manifolds were covered through the lens of topological- and smooth manifolds in Subsection 2.7.1, and lays the foundation for understanding the concept of Riemannian manifolds. Noted in [25], the metric properties of the Euclidean $\mathbb{R}^n$ are restricted to flat spaces. And hence, the Euclidean metric properties are not eligible to perform mathematical operations on the curved spaces of smooth manifolds.

Riemannian geometry studies smooth manifolds equipped with a Riemannian metric. From [25], a Riemannian metric on a smooth manifold $\mathcal{M}$ is a symmetric positive definite smooth 2-covariant tensor field g. As noted in [28], a smooth manifold $\mathcal{M}$ equipped with a Riemannian metric g is called a Riemannian manifold, and denoted by (M, g).

If g is a Riemannian metric on $\mathcal{M}$, then for each $x \in \mathcal{M}$, the 2-tensor g_x is an inner product on $T_x\mathcal{M}$. The notation of the inner product $\langle u, v\rangle_g$ denotes the real number $g_x(u, v)$ for $u, v \in T_x\mathcal{M}$ (Figure 2.5). The definition of a Riemannian metric allows for the usage of lengths, norms, angles and distances of a tangent vector $v \in T_x\mathcal{M}$. The length or norm of a tangent vector $v \in T_x\mathcal{M}$ is expressed in [28] as

$$|v|_g = \langle v, v\rangle_g^{1/2} = g_x(v, v)^{1/2}. \tag{2.42}$$

The angle between two nonzero tangent vectors $u, v \in T_xM$ is the unique $\theta \in [0, \pi]$ satisfying

$$\cos\theta = \frac{\langle u, v\rangle_g}{|u|_g |v|_g} \tag{2.43}$$

Tangent vectors $u, v \in T_xM$ are said to be orthogonal if $\langle u, v\rangle_g = 0$. This means either one or both vectors are zero, or the angle between them is $\pi/2$.

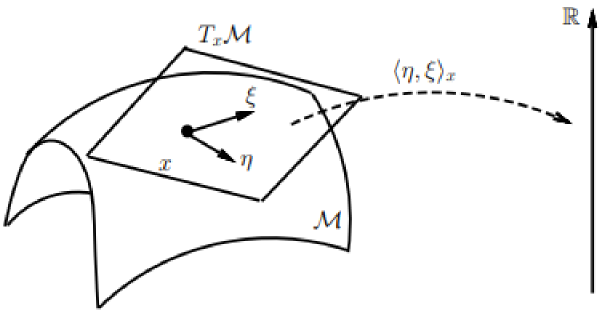


Figure 2.5.: The Riemannian metric with an inner product on a manifold.

2.7.3. Riemannian metric on $SO(3)$

The following is from [29]. The Riemannian metric on T_xSO_3 is expressed as

$$\langle A, B\rangle_g = \frac{1}{2}\operatorname{tr}\left(A^{\mathrm{T}}B\right), \quad A, B \in T_xSO_3. \tag{2.44}$$

The Riemannian metric of the two elements $u^\times$ and $v^\times$ on the Lie algebra $\mathfrak{so}(3)$ satisfies

$$\left\langle u^\times, v^\times \right\rangle_g = u^{\mathrm{T}}v, \tag{2.45}$$

which follows from the calculation

$$\langle u^\times, v^\times \rangle_g = \frac{1}{2} \operatorname{tr}\left[\left(u^\times\right)^{\mathrm{T}} v^\times \right] = -\frac{1}{2} \operatorname{tr}\left[u^\times v^\times \right], \tag{2.46}$$

which is equal to

$$\frac{1}{2} \operatorname{tr}\left(u^{\mathrm{T}} v I - u v^{\mathrm{T}} \right) = u^{\mathrm{T}} v. \tag{2.47}$$

Let R be a rotation matrix. The Riemannian metric of $Ru^\times, Rv^\times \in T_x SO(3)$ is

$$\langle Ru^\times, Rv^\times \rangle_g = \frac{1}{2} \operatorname{tr}\left[\left(u^\times\right)^{\mathrm{T}} R^{\mathrm{T}} R v^\times \right] = \frac{1}{2} \operatorname{tr}\left[\left(u^\times\right)^{\mathrm{T}} v^\times \right] = \langle u^\times, v^\times \rangle_g, \tag{2.48}$$

which shows that the Riemannian metric on $SO(3)$ is left invariant, as it is indifferent whether $u^\times$ and $v^\times$ are pre-multiplied by R. It is further shown that the Riemannian metric is right-invariant in $u^\times R, v^\times R \in T_x SO(3)$. The calculation gives

$$\langle u^\times R, v^\times R \rangle_g = \frac{1}{2} \operatorname{tr}\left[R^{\mathrm{T}} \left(u^\times\right)^{\mathrm{T}} v^\times R \right] = \frac{1}{2} \operatorname{tr}\left[\left(u^\times\right)^{\mathrm{T}} v^\times \right] = \langle u^\times, v^\times \rangle_g. \tag{2.49}$$

Being both left- and right-invariant means that the Riemannian metric on $SO(3)$ is bi-invariant [30], as it is unchanged whether $u^\times$ and $v^\times$ are pre- or post-multiplied by the rotation R. Bi-invariance means that the distance between two points points are unaltered if both points are given the same offset.

The Riemannian metric on $SO(3)$ makes it possible to perform mathematical operations on the tangent space, through surjective mappings, which permits movement along a geodesic curve γ between two points (x_1, x_2) on $\mathcal{M}$. The geodesic curve γ denoted as $d_{\mathcal{M}}$ is defined as the infimum length between two distinct points on $\mathcal{M}$, i.e the shortest path between two points. The geodesic distance is seen to be the angular distance defined in Subsubsection 2.2.2. The angular distance induced by the Riemannian metric is further elaborated on in Subsubsection 2.7.3. Figure 2.6 depicts the movement along γ on $\mathcal{M}$ from point C to C_i. It is seen from the figure that the logarithm- and exponential map from Equation 2.3-Equation 2.6 allows for mapping between $T_x SO(3)$ and $SO(3)$.

$$d_{\mathcal{M}} = \left\| \log(C^{-1}C_i) \right\|_F^2 \tag{2.50}$$

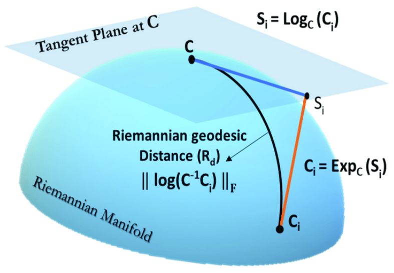


Figure 2.6.: Geodesic on the Riemannian manifold $SO(n)$.

Geodesic distance $d_{\mathcal{M}}$ on $SO(3)$ on Riemannian manifolds

Consider the motion from $R \in SO(3)$ to $Q \in SO(3)$ described by the rotation with angular velocity $\omega(t) = \omega k$, for $0 \le t \le T$, where ω is constant, and k is a constant unit vector. Moreover, it is assumed that $Q = R\exp(\theta k)$, which means that $\omega T = \theta$. This further leads to

$$R(t) = R\exp(\omega t k), \quad 0 \le t \le T \tag{2.51}$$

The Riemannian metric is then given in [29][15] as

$$\left\langle \omega^{\times}, \omega^{\times} \right\rangle_g = \omega^{\mathrm{T}}\omega = \omega^2 k^{\mathrm{T}} k = \omega^2 \tag{2.52}$$

The length of a curve induced by the Riemannian metric is then

$$d_{\mathcal{M}} = \int_{t=0}^{T} \sqrt{\left\langle \omega^{\times}, \omega^{\times} \right\rangle_g} \mathrm{d}t, \tag{2.53}$$

which gives

$$\int_{t=0}^{T} \omega \mathrm{d}t = \omega T = \theta. \tag{2.54}$$

This proves that the length given by the Riemannian metric is the angular distance, which is termed as the geodesic distance $d_{\mathcal{M}}$. The geodesic on a Riemannian manifold $(\mathcal{M}, g)$ is expressed as

$$dist : \mathcal{M} \times \mathcal{M} \to \mathbb{R} : dist(x, y) = \inf_{\Gamma} d_{\mathcal{M}}, \tag{2.55}$$

where Γ is the set of all such curves in $\mathcal{M}$ which connects points x and y in which the geodesic is given as the infimum length between two points [28].

2.8. Optimization

2.8.1. Euclidean optimization

Before delving into Riemannian optimization, a brief summary of Euclidean optimization must be introduced. Let $\mathbb{R}^n$ be the Euclidean space and let $f : \mathbb{R}^n \to \mathbb{R}$ be a real-valued function. An optimization problem on this space has the form

$$\arg\min_{x \in \mathbb{R}^n} f(x) \tag{2.56}$$

The equation states that one would like to find a point $\hat{x} \in \mathbb{R}^n$ such that $f(\hat{x})$ is the minimum of f. The optimization problem derives the minimum with the use of Euclidean gradients. The function $f(x) = \frac{1}{2}\left(x^2 + y^2 + z^2\right) = \frac{1}{2}x^{\mathrm{T}}x$ will have the Euclidean gradient as

$$\nabla f(x) = \begin{bmatrix} x & y & z \end{bmatrix}^{\mathrm{T}} = x \tag{2.57}$$

The numerical method for solving Equation 2.56 is given by the stochastic gradient descent algorithm as

Algorithm 1 Stochastic Gradient Descent

```
1. Pick arbitrary $x_{(0)} \in \mathbb{R}^n$ and let $\alpha \in \mathbb{R}$ with $\alpha > 0$
2. While the stopping criterion is not satisfied:
   1. Compute the gradient of $f$ at $x_{(t)}$, i.e. $h_{(t)} := \nabla f(x_t)$
   2. Move in the direction of $-h_{(t)}$, i.e. $x_{(t+1)} = x_{(t)} - \alpha h_{(t)}$
   3. $t = t + 1$
3. Return $x_{(t)}$
```

2.8.2. Riemannian optimization

The gradient descent algorithm can be generalized on Riemannian manifolds with Riemannian gradients. Consider $(\mathcal{M}, g)$ to be an n-dimensional Riemannian manifold. The union of all tangent spaces on $\mathcal{M}$ defines the tangent bundle $\mathcal{TM} = \cup_{\mathbf{x}\in\mathcal{M}} T_x\mathcal{M}$. Let $f : \mathcal{M} \to \mathbb{R}$ be a real-valued function on $\mathcal{M}$ and $\forall(x, \eta) \in \mathcal{TM}$. The tangent bundle defines a vector field on $\mathcal{M}$. The Riemannian optimization problem on $\mathcal{M}$ is given in simple form as

$$\arg\min_{x\in\mathcal{M}} f(x). \tag{2.58}$$

Consider $\eta \in \mathcal{M}$ to be the tangent vector at $T_x\mathcal{M}$ if there exists a geodesic curve $\gamma : [0, 1]$ on $\mathcal{M}$. It follows in [1] that $\gamma(0) = x$ and the time-derivative $\dot{\gamma}(0) = \eta$. The Riemannian gradient of f on $\mathcal{M}$ is thus a unique tangent vector $\tilde{\nabla} f$ in the vector field defined on $\mathcal{M}$, and satisfies the directional derivative as

$$Df(x)[\eta] = \langle \tilde{\nabla} f(x), \eta \rangle_g \tag{2.59}$$

where $\mathrm{D}f(x)[\eta]$ is the derivation of f by η. The Riemannian gradient of f at x is the direction in which the directional derivative is the greatest (steepest). The Riemannian gradient descent (RGD) is given in [1] by

$$R_{x_{k+1}} \leftarrow R_{x_k}\left(-\tau_k \tilde{\nabla} f_{(x_k)}\right), \tag{2.60}$$

where k is the iteration, τ_k is step size, grad $f(x_k)$ is the Riemannian gradient and R_{x_k} is the retraction. A retraction is a parameterization $R_k : T_x\mathcal{M} \to \mathcal{M}$, and is used to map x to the endpoint of the geodesic when $t = \mathrm{T}$ in Equation 2.51. The retraction on $SO(3)$ is simply the Rodrigues' equation, and satisfies the following

- R_x is continuously differentiable
- $R_x(0) = x$
- $\mathrm{D}R_x(0)[\eta] = \eta$

The retraction on $SO(3)$ is simply the Rodrigues' equation. A step along a geodesic curve with a retraction is depicted in Figure 2.7 [1].

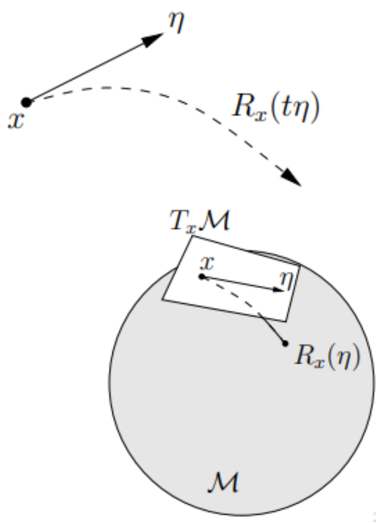


Figure 2.7.: Riemannian optimization on $(\mathcal{M}, g)$. η is tangent vector at $T_x\mathcal{M}$.

2.8.3. Riemannian optimization on $SO(3)$

Riemannian optimization on $SO(3)$ in the following is from [15]. The time derivative of the rotation matrix is in [12] given by

$$\dot{R} = R\omega_b^\times \in T_R SO(3), \tag{2.61}$$

where $\omega_b^\times = R^\mathrm{T}\dot{R}$ and $T_R SO(3)$ is the tangent space of $SO(3)$ at R. The tangent space at the identity $R = I$ is $T_I SO(3) = \mathfrak{so}(3)$, which verifies

$$\dot{R}\Big|_{R=I} = \omega_b^\times \in \mathfrak{so}(3). \tag{2.62}$$

Consider the Frobenius of two rotation matrices define the loss function as

$$\mathcal{L}(f(R)) = \|R_{est} - R_{gt}\|_F^2 \,. \tag{2.63}$$

Then $\mathcal{L}(f(R)) \in \mathbb{R}$ maps a rotation matrix $R \in SO(3)$ to a scalar $\mathbb{R}$. The gradient $\tilde{\nabla}\mathcal{L}$ of the loss function lies on the tangent plane at R, which is written as $\tilde{\nabla}\mathcal{L} \in T_x SO(3)$. The gradient can be expressed as

$$\tilde{\nabla}\mathcal{L} = Rg^\times \in T_R SO(3), \tag{2.64}$$

where $g^{\times} \in \mathfrak{so}(3)$. The directional derivative of the function $\mathcal{L}(f(R))$ is found by differentiating the function $\mathcal{L}(f(P(t)))$, where

$$P(t) = R \exp\left(ta^{\times}\right) \in SO(3). \tag{2.65}$$

Then $P(0) = R$, and

$$\dot{P}(0) = P(0)a^{\times} = Ra^{\times}. \tag{2.66}$$

Moreover, $\dot{P}(0) = P(0)\omega(0)^{\times}$ where $\omega(t)^{\times} = P^{\mathrm{T}}\dot{P}$ is the right velocity corresponding to $P(t)$. From this it is seen that $\omega(0) = a$. The gradient at R is then defined in terms of the directional derivative and the Riemannian metric by

$$\left\langle Ra^{\times}, \tilde{\nabla}\mathcal{L} \right\rangle_g = \frac{\mathrm{d}}{\mathrm{d}t} f(P(t)) \bigg|_{t=0}. \tag{2.67}$$

Since the Riemannian metric is bi-invariant on $SO(3)$, the gradient can be alternatively expressed as

$$\left\langle a^{\times}, g^{\times} \right\rangle_g = \frac{\mathrm{d}}{\mathrm{d}t} f(P(t)) \bigg|_{t=0} \tag{2.68}$$

The Riemannian optimization problem on $SO(3)$ can be expressed as

$$R_{x_{k+1}} \leftarrow R_{x_k}\left(-\tau_k \tilde{\nabla}\mathcal{L}_{(x_k)})\right), \tag{2.69}$$

where τ_k is the step size, $\tilde{\nabla}\mathcal{L}$ is the Riemannian gradient, R_{x_k} is the retraction and k is the iteration k.

Chapter 3.

Deep Learning on Point Clouds

This chapter introduces point cloud registration with deep learning, and the challenges of using deep learning on point clouds. In particular, the chapter reviews the applied CNN-architectures in [5], [6], [7], [8] and [1] for conducting deep rotation regression using PointNet and PointNet++. It is assumed that the reader is familiar with the concept of deep learning and the inner-workings of a CNN.

3.1. Pose estimation and loss function

Consider the data point cloud $\mathcal{X}$ and the model point cloud $\mathcal{Y}$, where $\mathcal{X} = [x_1, \ldots, x_N] \in \mathbb{R}^{3\times N}$ and $\mathcal{Y} = [y_1, \ldots, y_N] \in \mathbb{R}^{3\times N}$ where N is the number of points, each point is a 3D vector and each pair (x_i, y_i) is a point correspondence [6]. The point clouds are assumed to be separated by a rotation R.

$$y_i = Rx_i \tag{3.1}$$

If the registration problem involves the estimation of a rotation R between the data- and model point cloud, the problem is generally known as the *Wahba's-problem* [31]. The loss function of the Wahba's problem is generally computed by formulating it as a least-squares problem

$$\underset{R\in SO_3}{\arg\min} \sum_{i=1}^{N} \left\| y_i - Rx_i \right\|^2 , \tag{3.2}$$

which minimizes the sum of the squared differences between model- and target point cloud. As seen in the equation, the least-squares minimization is given by the ℓ_2-norm. Thus, the minimization problem does not regress directly on the rotation matrices, but uses the i-th vector in both data sets to find the minimal solution. This is in contrast to deep rotation regression which regresses directly on the rotation matrices under supervised learning. The loss function $\mathcal{L}(f(R))$ in the backpropagation of a neural network is given in [1] by the Frobenius norm as

$$\underset{R \in SO_3}{\arg\min} \, \|R_{est} - R_{gt}\|_F^2 \,, \tag{3.3}$$

where f constructs a loss function that compares the estimated rotation R_{est} to the ground truth rotation R_{gt}.

3.2. Deep learning on point clouds

The application of deep learning on point clouds imposes multiple challenges, where the most obvious difficulties could be distinguished into irregularities, unstructuredness and unorderedness.

Irregularity: Point clouds are irregular, which means that points of an object/scene are not evenly sampled, as some regions are more dense of points, whereas other areas are more sparse [32]. Figure 3.1 illustrates the concept of irregularities on a car model.

Unstructured: Point clouds are not on a regular grid, which means that each point is scanned independently and its distance to neighboring points is not always fixed, whereas pixels in images are fixed on a 2-dimensional grid with fixed spacing between each pixel [32]. Figure 3.2 illustrates the concept of unstructuredness.

Unordered: The order of the points in a point cloud data set does not change the scene the points are representing [32]. Figure 3.3 shows the

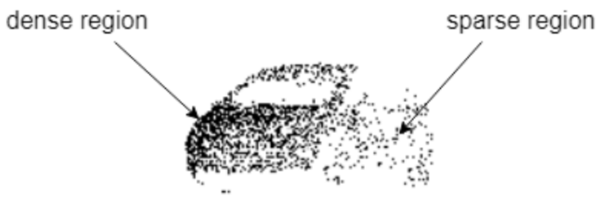


Figure 3.1.: Irregularities of points on a car model produces dense and sparse areas of points. Figure is from [32].

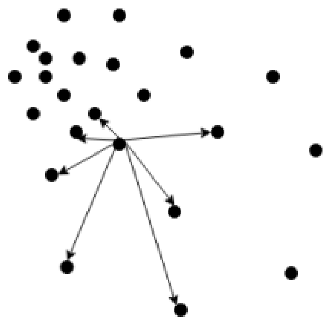


Figure 3.2.: A point cloud is unstructured. Thus it has no grid, as each point is independent and distance between neighboring points is not fixed. Figure is from [32].

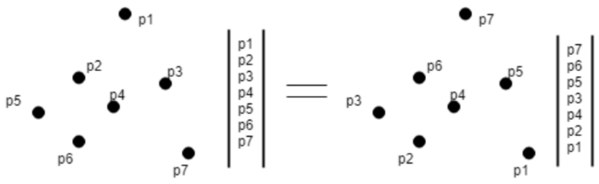


Figure 3.3.: Point clouds are invariant to permutations. Figure is from [32].

3.2.1. PointNet

PointNet is the first deep learning framework on unstructured point clouds, and is a bedrock for most of the later developed frameworks such as PointNet++ [32]. PointNet is a unified weight-sharing CNN model developed for 3D shape segmentation and classification purposes using raw a point cloud as input. Unlike pixel arrays in image classification tasks, a fundamental problem lies in the fact that point clouds are unordered. Given that a point cloud $\mathcal{X} \in \mathbb{R}^{3\times N}$ is an unordered data set, the network must be invariant to $N!$ permutations of the data set. PointNet obtains permutation invariance, and the classification architecture of the network is given in Figure 3.4.

The idea of PointNet is to learn a spatial encoding of each point through a multi-layer perceptrons (MLPs) and then aggregate all individual point features to a global point cloud signature using max-pooling [34]. The diagram above illustrates intuitively the inner-workings and the pipeline of PointNet. Given an unordered point set $\mathcal{X} = [x_1, \dots, x_N] \in \mathbb{R}^{3\times N}$, one can define a set function $f : \mathcal{X} \to \mathbb{R}$ that maps a set of points to a vector

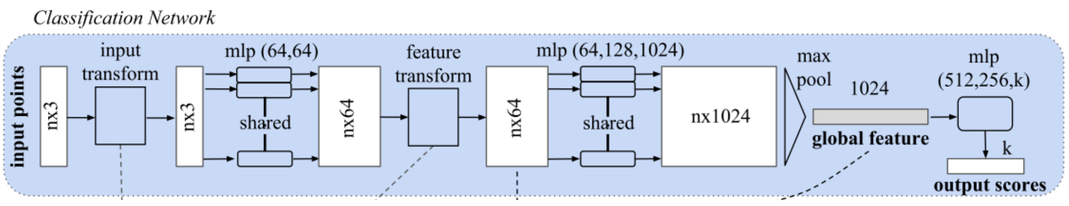


Figure 3.4.: PointNet. Figure is from [33].

$$f\left(x_1,\ldots,x_N\right)=\gamma\left(\underset{i=1,\ldots,n}{\text{MAX}}\,\psi\left(x_i\right)\right) \tag{3.4}$$

where γ and ψ MLPs. f in Equation 3.2.1 is permutation invariant, and the MAX is a max pooling operator that takes a data set of n vectors as input and returns a vector of the element-wise maximum [34]. Permutation invariance is achieved by processing all points independently in shared MLPs which creates shared weights [5]. The classification network is composed of two weight-sharing MLPs. ψ is a feature extractor with neuron sizes of [64,64,64,128,1024] where all input points in $\mathcal{X}$ share a single copy of ψ [33]. The neural network maps the point cloud to $\tilde{\mathcal{X}} = \Psi(\mathcal{X})$ such that $\tilde{\mathcal{X}} \in \mathbb{R}^{1024\times N}$. $\tilde{\mathcal{X}}$ is then further processed through max pooling to create 1024D global feature vector. Finally, the feature vector is then passed through the second MLP, γ with output sizes of [512, 256, n], resulting in a n-dimensional output vector $\mathbb{R}^k$.

3.2.2. PointNet++

Recall that $\mathcal{X} \in \mathbb{R}^{3\times N}$ is a point cloud. All points in the point cloud forms local dependency/structure with their surrounding points [33]. Capturing the local structure has proven to be essential for the success of CNN-architectures [33]. The PointNet introduced in Subsection 3.2.1 does not consider the local structure of each individual point, which imposes shortcomings in recognizing fine grained patterns in the input set, which further leads to restricted abilities of generalization of complex scenes [34]. After PointNet many approaches were proposed to capture local structure. PointNet++, developed by [34], is one such proposal which is PointNet with a local structure added hierarchically, with each hierarchy encoding a richer representation [32]. The addition of a hierarchical structure shows to overall improve the performance in classification tasks [34]. The hierarchical neural network applies PointNet recursively on a nested partitioning of the input set $\mathcal{X}$, and by exploiting metric space distances, the network is able to learn the local- and higher level features. This process resembles CNN for image classification, where the convolutional layers extracts local spatial features from the image and combines the local spatial features to higher-order features. The higher-order features are then used to linearly separate different image types [35]. Figure 3.5 illustrates the architecture of PointNet++ with its hierarchical structure. The grey shaded area to the left in the diagram shows the hierarchical structure.

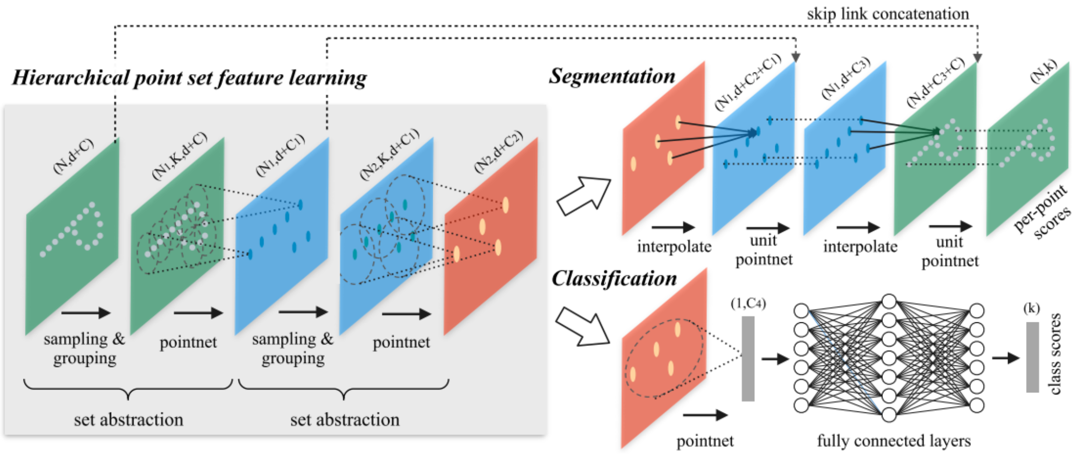


Figure 3.5.: PointNet++. Figure is from [34].

Local structure modeling rests on three operations: *sampling layer*, *grouping layer* and a mapping function (MLP) [32]. As seen in the Figure 3.5, the hierarchical structure is formed by several *set abstraction* levels, where a set of points at each level is processed and abstracted to produce a new set with fewer elements [34]. Each set abstraction level is composed of a sampling layer, grouping layer and PointNet layer.

Sampling layer

The Sampling layer is applied to reduce resolution of points across layers. a set of points from the input set, which defines the centroids of local regions. Given point cloud $\mathcal{X} \in \mathbb{R}^{3\times N}$, the sampling reduces it to M points $\hat{\mathcal{X}} \in \mathbb{R}^{3\times M}$, where $M < N$. The subsampled M points are referred to as centroids. The centroids are used to represent the local region from which they were sampled [32]. There most prominent techniques for subsampling are:

- Random Point Sampling (RPS) where each of the N points is uniformly likely to be sampled.

- Farthst Point Sampling (FPS) where the M sampled points is the most distant point from the rest of the $M - 1$ points.

Grouping layer

Given that the centroids are sampled, k-Nearest Neighbor-algorithm (kNN) is used to form local patches by grouping centroid points with their nearest neighboring points. The points in a local patch are then used to compute the local feature representation of the neighborhood. In the grouping layer, the kNN-algorithm is either used explicitly where the k-nearest neighbors are sampled to form a local path, or in a ball-query, where a ball-query selects the k-nearest neighbor points within a given radius [32].

PointNet layer

Given that the nearest points to each centroid are computed, the next stage is to map the points into a global feature vector. This is procedure is executed by applying Equation Equation 3.2.1 [32].

Figure 3.6 depicts the process of local structure modeling using an airplane model as point cloud input.

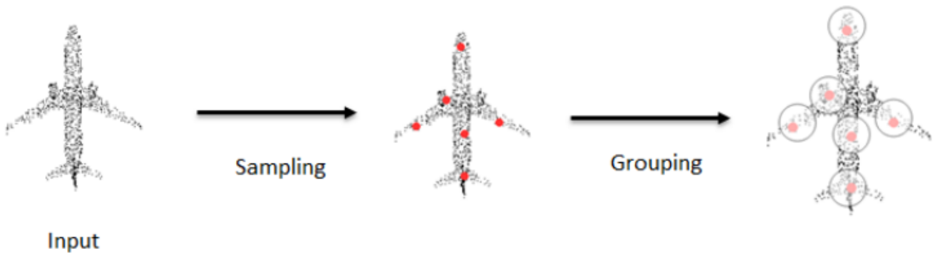


Figure 3.6.: Sampling and grouping of points into local patch. The reds are the centroid points selected using sampling algorithms, and the grouping shown is a ball query where points are selected based on a radius distance to the centroid [32]. Figure is from [32].

3.2.3. PointNet++ MSG:

Recall that point clouds are irregular. Features learned in denser data, does not necessarily generalize well to sparsely sampled regions. Moreover, PointNet++ trained on sparse input sets, does not necessarily learn local structures well enough. To tackle this problem, [34] provides PointNet++ with density adaptive layers, which is called *Multi-scale grouping* (MSG). Figure 3.7 illustrates PointNet++ MSG with its adaptive layers, that learn to combine features from regions of different scales when the sampling density changes.

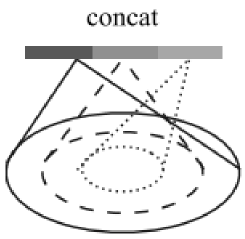


Figure 3.7.: PointNet++ MSG. Figure is from [34].

3.3. Deep rotation regression

To regress rotations with the PointNet and PointNet++, one must obtain the R_{est} from Equation 3.3. The solution is setting the neural network output dimension to be equal to the desired rotation representation in $\mathbb{R}^n$, which is parameterized to a rotation matrix $R \in SO(3)$ with the parameterizations introduced in Section 2.5.

3.3.1. PointNet

The contributions from Zhou et al.[6], Levinson et al.[7] and Brègier [8] are conducted by using PointNet as the backbone network. The PointNet-architecture in [6] receives two input point clouds $\mathcal{X} \in \mathbb{R}^{3\times N}$ and $\mathcal{Y} \in \mathbb{R}^{3\times N}$, where $\mathcal{Y} = R_{gt}\mathcal{X}$. The two input point clouds are pushed through PointNet to generate R_{est}, which is used to construct the loss against R_{gt} in Equation 3.3. Recall ψ and γ to be two weight-sharing MLPs from Equation 3.2.1. In particular, consider two input point clouds $\mathcal{X} \in \mathbb{R}^{3\times N}$ and $\mathcal{Y} \in \mathbb{R}^{3\times N}$, both point clouds are separately passed through ψ to create $\tilde{\mathcal{X}} = \psi(\mathcal{X})$ and $\tilde{\mathcal{Y}} = \psi(\mathcal{Y})$. Both $\tilde{\mathcal{X}} \in \mathbb{R}^{1024\times N}$ and $\tilde{\mathcal{Y}} \in \mathbb{R}^{1024\times N}$ are then concatenated to form a $\mathcal{Z} \in \mathbb{R}^{2048\times N}$. $\mathcal{Z}$ is then passed through the γ with output sizes of [2048, 512, n]. The following code snippet is from [6], and shows the MLPs' ψ and γ. The code is written in Python and uses the PyTorch framework.

```python

import torch
import torch.nn as nn


"""Feature descriptor"""
self.feature_extracter = nn.Sequential(
        nn.Conv1d(3, 64, kernel_size=1),
        nn.LeakyReLU(),
        nn.Conv1d(64, 128, kernel_size=1),
        nn.LeakyReLU(),
        nn.Conv1d(128, 1024, kernel_size=1),
        nn.AdaptiveMaxPool1d(output_size=1))



"""Multilayer perceptron"""
self.mlp = nn.Sequential(
        nn.Linear(2048, 512),
        nn.LeakyReLU(),
        nn.Linear(512, self.out_channel))

#self.out_channel = D-dimensional output

"""Two input point clouds pt1 and pt2"""
def forward(self, pt1, pt2):
        batch = pt1.shape[0]
        point_num =pt1.shape[1]

        feature_pt1 = self.feature_extracter(pt1.transpose(1,2)).
                                view(batch,-1)#b*512
        feature_pt2 = self.feature_extracter(pt2.transpose(1,2)).
                                view(batch,-1)#b*512

        f = torch.cat((feature_pt1, feature_pt2), 1) #batch*1024
```

3.3.2. PointNet++ MSG

In Chen et al.[1] the PointNet++ MSG is used as the backbone network for regressing rotations. The network receives a single point cloud as input to generate R_{est}, which is used to form a loss against R_{gt}. Thus, the network and the formulation of the regression problem in [1] is in contrast to the previous works mentioned in Subsection 3.3.1. The following code snippet is from [1], and shows PointNet++ MSG. The code is written in Python and uses the PyTorch framework.

```
class PointNet2_MSG(nn.Module):
    def __init__(self, out_channel):
        super(PointNet2_MSG, self).__init__()
        self.sa1 = PointNetSetAbstractionMsg(512, [0.1, 0.2, 0.4],
                                                   [32, 64, 128], 3,
                                                   [[32, 32, 64],
                                                   [64, 64, 128],
                                                   [64, 96, 128]])
        self.sa2 = PointNetSetAbstractionMsg(128,
                                             [0.4,0.8],
                                             [64, 128],
                                             128+128+64,
                                             [[128, 128, 256],
                                             [128, 196, 256]])

        self.sa3 = PointNetSetAbstraction(npoint=None, radius=None,
                                  nsample=None, in_channel=512 + 3,
                                  mlp=[256, 512, 1024], group_all=
                                  True)

        self.mlp = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(),
            nn.Linear(512, out_channel))

    def forward(self, xyz):
        # Set Abstraction layers
        B,C,N = xyz.shape
        l0_points = xyz
        l0_xyz = xyz
        l1_xyz, l1_points = self.sa1(l0_xyz, l0_points)
        l2_xyz, l2_points = self.sa2(l1_xyz, l1_points)
        l3_xyz, l3_points = self.sa3(l2_xyz, l2_points)

        out_data = self.mlp(l3_points.squeeze(-1))
        return out_data
```

Chapter 4.

Deep Rotation Regression

4.1. Problem area

Gao et al.[5] initiated the era of deep rotation regression by directly regressing on rotation matrices constructed from point cloud feature vectors in $\mathbb{R}^3$, by using the axis-angle parameterization (check Section 2.5 for details) to form the rotation matrix. Figure 4.1 shows a diagram of how PointNet was used in [5] to generate $r \in \mathbb{R}^3$. Note that the input dimension is $\mathbb{R}^{6\times N}$, as each point has 6 dimension: 3 dimensions for spatial coordinates and 3 dimensions for color information (RGB) [5].

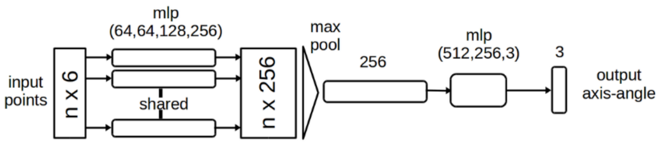


Figure 4.1.: PointNet with 3-dimensional output. Figure is from [5].

The work of Gao et al.[5] has since been extended in Zhou et al.[6], Peretroukhin et al.[19], Levinson et al.[7] and Brègier [8]. Recall the parameterization to be the mapping from an n-dimensional network output to a rotation matrix R_{est}. A great challenge in deep rotation regression is to construct learning friendly rotation representations for network training. It is seen that when the full rotation space is required ($\theta = [0, 2\pi]$), the network generates provably wrong results for certain parameterizations, which was revealed by Zhou et al.[6] to be caused by discontinuities. The root of discontinuities is related to the topological concepts introduced in Subsection 2.6.2 about homeomorphism between the rotation space $SO(3)$ and $\mathbb{R}^n$. The discontinuities are limited to 3D and 4D rotation representations, which includes the traditionally used quaternions, Euler angles and axis-angles.

Considering the fact that most neural networks are continuous, which allows for gradient based optimization, discontinuities imposed by rotation representations generates a negative impact on neural network learning [36]. As already known from Chapter 2, rotations reside in the non-Euclidean manifold of $SO(n)$, while the neural network outputs from both PointNet and PointNet++ are nested in $\mathbb{R}^n$. Zhou et al.[6] proved that the discontinuities are enforced because there are no homeomorphic embeddings between $\mathbb{R}^n$ and the rotation space $SO(3)$, when $n < 5$.

[6] proposed parameterization through Gram-Schmidt orthogonalization using 6D representation and 5D representations. [19] proposed 10D representations, while [7] proposed a 9D representation and forming the rotation matrix through SVD-orthogonalization. A recent paper from Chen et al.[1] hypothesises that naively using Euclidean gradients during backpropagation, usually leads to a new matrix off $SO(3)$ manifold. The off-manifold components will lead to noise in the gradients of the neural network weights, which will further harm generalization and convergence [1]. The contribution in [1] offers *manifold-aware gradients*, which leverages from Riemannian optimization from Section 2.8 to find the best possible gradients for backpropagation into the network weights. Thus, the common objective in all of these promising aforementioned contributions, is narrowing the gap between $\mathbb{R}^n$ and $SO(3)$ manifold, as the desired state is to perform regression on $SO(3)$ without discontinuities.

4.2. Continuity of rotation representations

This section covers the contributions from Zhou et al.[6] and Brègier [8] on the topic on learning-friendly rotation representations.

4.2.1. Deep learning pipeline

The deep learning pipeline consists of a forward-and backward pass. In the forward pass, the neural network outputs a raw n-dimensional vector x in a Euclidean space (ambient space) $\mathcal{X} = \mathbb{R}^n$. Then the manifold mapping π maps x to $\hat{x} = \pi(x) \in \mathcal{M}$, followed by a rotation mapping $\phi(\pi(\hat{x}))$ onto the rotation manifold $SO(3)$, such that the optimization variable is regressed on $SO(3)$. The inverse mapping is then a map back to $\mathcal{M}$ by ψ.

For network outputs $x \in \mathbb{R}^3$, the manifold mapping $\pi(x)$ is not required, as a Euclidean neural network can output 3D vectors [1]. However, for dimensions $n > 3$, the n-dimensional vector lies on a non-Euclidean manifold. A manifold mapping by the form of a normalization/orthogonalization step onto the manifold $\pi : \mathbb{R}^n \to \mathcal{M}$ is required [1], such that the output further ends up in the rotation space $SO(3)$ in the rotation mapping ϕ. Thus, for 4D/10D, 5D/6D and 9D, the representation mapping induced by π maps the mentioned representations to $S^3, \mathcal{V}_2\left(R^3\right)$ and $SO(3)$, respectively [1]. Note that representation- and rotation mapping for 9D output is the identity. Figure 4.2 shows the pipeline in a simpler form.

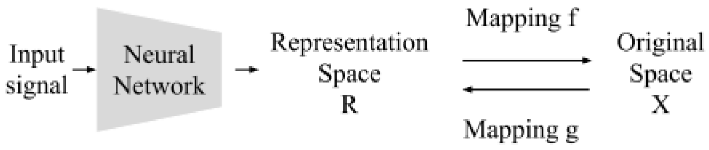


Figure 4.2.: Pipeline with input, output and mapping between the representation space and the original space. Figure is from [6].

4.2.2. Smoothness properties & surjectivity

Results for continuous functions indicate that functions that have better smoothness properties have lower approximation error [6]. The authors in [6] stated that ψ and ϕ must be continuous in order for the network to be continuous at all times. The choice of a mapping function and n-dimensional representation is critical to ensure learning-friendly neural network training.

Discontinuity

Let $\theta \in R$ be the rotation angle, and $R = [0, 2\pi]$ a suitable set of angles. Consider ψ be a mapping function from $SO(2)$ to the representation space R, then ψ imposes a discontinuous map at the identity rotation at $\theta = 0$ and 2π. It is noted in [36] that neural networks confronts an obstacle when converting rotation matrices to quaternions and Euler angles, and produces a geodesic error ($d_{\mathcal{M}}$) of π radians for some input. Figure 4.3 depicts discontinuities during the inverse mapping from $SO(2)$ (Original Space) to $\mathcal{M}$ (Representation Space). The inverse mapping ψ in Figure 4.3 is given as g.

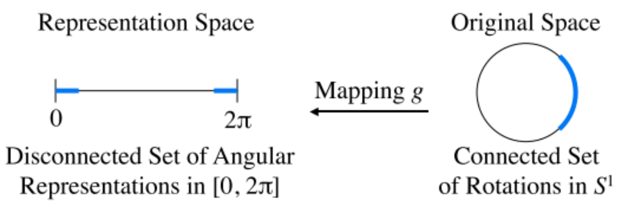

Representation Space
Original Space
Mapping g
0
2π
Disconnected Set of Angular Representations in [0, 2π]
Connected Set of Rotations in S¹

Figure 4.3.: Discontinuity. Figure is from [6].

The discontinuities imposed when mapping from $SO(3)$ to quaternions have been discussed in [36]. Let R be a rotation matrix. If $\mathrm{tr}(R) > -1$, the representation mapping $\psi_{4D} : R \in SO(3) \to q \in S^3$ is noted to be

$$\psi(R) = \left(\frac{\gamma}{2}, \frac{1}{2\gamma}(R_{32} - R_{23}), \frac{1}{2\gamma}(R_{13} - R_{31}), \frac{1}{2\gamma}(R_{21} - R_{12})\right) \tag{4.1}$$

where $\gamma = \sqrt{1 + \mathrm{tr}(R)}$. Since quaternions q and $-q$ identifies the same rotation, any conversion from R to quaternion needs to break ties. The conversion given Equation 4.1 must break ties towards the first coordinate being positive. Consider $R_z(\gamma) : [0, 1] \to \mathrm{SO}(3)$ defined by

$$R_z(\gamma) = \begin{bmatrix} \cos 2\pi\gamma & -\sin 2\pi\gamma & 0 \\ \sin 2\pi\gamma & \cos 2\pi\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.2}$$

where $R_z(\gamma)$ as the rotation around z-axis by angle $2\pi\gamma$. Then $\psi(R_z(\gamma)) = (\cos\pi\gamma, 0, 0, \sin\pi\gamma)$ when $R_z \in \left[0, \frac{1}{2}\right)$ and $\psi(R_z(\gamma)) = (-\cos\pi\gamma, 0, 0, -\sin\pi\gamma)$ when $R_z \in \left(\frac{1}{2}, 1\right]$. This gives

$$\lim_{\gamma \to \frac{1}{2}^-} \psi(R_z(\gamma)) = (0, 0, 0, 1) \neq (0, 0, 0, -1) = \lim_{\gamma \to \frac{1}{2}^+} \psi(R_z(\gamma)). \tag{4.3}$$

Thus ψ is not continuous at $\psi\left(\frac{1}{2}\right)$. Since neural networks typically compute continuous functions, such a function cannot be computed by a neural network [36]. It is then seen in [6] that for any continuous function $\psi_{4D} : SO(3) \to S^3$, there exists a rotation $R \in SO(3)$ such that the geodesic distance gives $d_{\mathcal{M}}(R_1, R_2) = \pi$.

Continuity

Zhou et al.[6] proposed that in order for a mapping function to be suitable for deep learning applications, the parameterization ϕ should be surjective and satisfy a notion of continuity, such that the right inverse $\psi : SO(3) \to \mathbb{R}^n$ exists. It is further noted that if the rotation space $SO(3)$ is not homeomorphic to any subset of the $\mathbb{R}^n$, then there are no continuous representations. This concept was used by [6] to create mappings from matrices through 5D and 6D representations, by using the adapted Gram-Schmidt orthogonalization presented in Section 2.5. [8] considers the preposition of surjectivity from [6] as part of several other properties that must be fulfilled in order to generate a learning-friendly regression on manifold.

The notion of ϕ being surjective is required to be able to predict any arbitrary output $\phi(x) \in SO(3)$. [8] declared that the the space where the regression is held should be a smooth manifold. As introduced in Subsection 2.7.1, $SO(3)$ is a differentiable manifold. Moreover, Brègier proposed other desirable properties such as

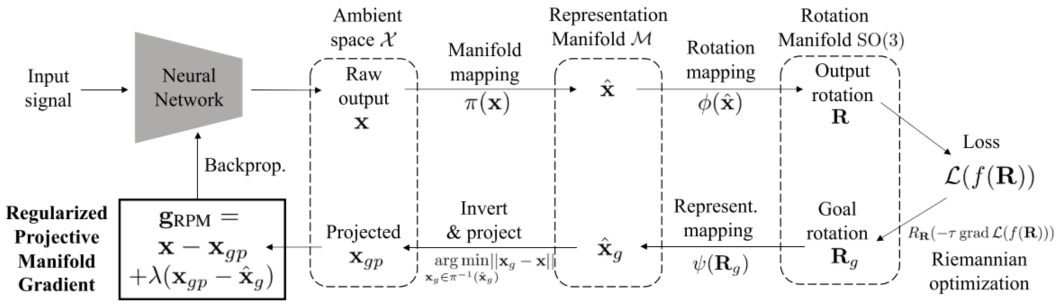
- **Jacobian of full rank**: The Jacobian of ϕ should be the rank of the dimension of $SO(3)$. This property ensures that one can always find an infinitesimal displacement to apply to x in order to achieve an arbitrary infinitesimal displacement of the output $\phi(x)$, such that there continuously exists an element to backpropagate during training. It is noted in [8] that a full rank Jacobian guarantees convergence of gradient descent towards a global minimum of $\hat{x}$.

- **Pre-images connectivity**: Recall the concept of pre-image connectivity introduced in Subsection 2.6.1. In [8] it is noted that bijective correspondences helps generalization in a neural network, which further leads to the notion of the existence of homeomorphism between output- and rotation space.

Thus, Brègier [8] laid the foundation for learning-friendly parameterizations onto $SO(3)$. Gathering the knowledge of satisfying conditions for learning-friendly parameterizations, one could then extrapolate which properties are fulfilled in the various parameterizations introduced in Section 2.5. Euler angles and axis-angles satisfies surjectivity. However, as their rotation representations are not homeomorphic to $SO(3)$, there are no pre-images connectivity due to many-to-one/one-to-many correspondences between. Moreover, both parameterizations do not fulfill a full rank Jacobian. It is noted in [8] that the axis-angle parameterization suffers from rank deficiency for input rotations of angles $2\pi k, k \in \mathbb{N}$. The axis-angle parameterization is suited for smaller angles [8]. The unit quaternion satisfies all but pre-images connectivity, while the 6D, 9D and 10D representations satisfies all conditions [8].

4.3. Manifold-aware gradients

Despite discovering learning-friendly rotation representations for network regression on the $SO(3)$ manifold, a newly published paper from Chen et al.[1] states that the regression step itself has been overlooked and neglected. The authors argue that by using Euclidean gradients derived from *vanilla auto-differentation* for backpropagation, will usually lead to a new matrix off $SO(3)$ manifold, which in turn will impose errors in the gradient of neural network weights. Solving this challenge involves applying geometric deep learning which generalizes the optimization problem onto non-Eulidean domains, i.e leveraging from Riemannian optimization. The idea of [1] is to construct an intermediate goal rotation R_g along the geodesic curve between R_{est} and R_{gt}, and use the goal rotation to find the gradient with the smallest norm. The gradient with the smallest norm is employed to update the output rotation to the goal rotation, and is denoted in [1] as a *manifold-aware gradient*. In particular, [1] introduces three manifold-aware gradients, denoted as g_M, g_{PM} and g_{RPM}. To find these gradients [1] introduces two new hyperparameters λ and τ, where tweaking λ in an interval from $[0, 1]$ determines the type of manifold-aware gradient, while τ determines the goal rotation. The manifold-aware gradients directly updates the neural network weights in the backpropagation in the backward pass. Hence, the forward pass in the pipeline presented in Figure 4.2 will remain unchanged. The modified backward pass in the pipeline is depicted in Figure 4.4. Note that the term RPMG-layer is not tied to a specific manifold-aware gradient, but is merely used as a generalization of the domain where g_M, g_{PM} and g_{RPM} are constructed.



Input signal
Neural Network
Ambient space $\mathcal{X}$
Raw output $\mathbf{x}$
Manifold mapping $\pi(\mathbf{x})$
Representation Manifold $\mathcal{M}$
$\hat{\mathbf{x}}$
Rotation mapping $\phi(\hat{\mathbf{x}})$
Rotation Manifold SO(3)
Output rotation $\mathbf{R}$
Loss
$\mathcal{L}(f(\mathbf{R}))$
Backprop.
Regularized Projective Manifold Gradient
$\mathbf{g}_{\text{RPM}} = \mathbf{x} - \mathbf{x}_{gp} + \lambda(\mathbf{x}_{gp} - \hat{\mathbf{x}}_g)$
Projected $\mathbf{x}_{gp}$
Invert & project
$\arg\min_{\mathbf{x}_g \in \pi^{-1}(\hat{\mathbf{x}}_g)} ||\mathbf{x}_g - \mathbf{x}||$
$\hat{\mathbf{x}}_g$
Represent. mapping $\psi(\mathbf{R}_g)$
Goal rotation $\mathbf{R}_g$
$R_{\mathbf{R}}(-\tau \operatorname{grad} \mathcal{L}(f(\mathbf{R})))$
Riemannian optimization

Figure 4.4.: Pipeline with RPMG. Figure is from [1].

4.3.1. Backpropagation with RPMG-layer

The better gradient with x^*

Noted in [1], consider the ℓ_2-loss to be a general regression problem in $\mathbb{R}^n$. The ℓ_2-loss is then given as

$$\arg\min \|x - x_{gt}\|^2, \tag{4.4}$$

where x is the network output, and x_{gt} is ground-truth. The gradient is then noted as

$$g = 2\left(x - x_{gt}\right) \tag{4.5}$$

Recall the Frobenius norm $\|R_{est} - R_{gt}\|_F^2$ from Equation 3.3 as a regression problem on $SO(3)$. Using the notion of g in Equation 4.5, the authors in [1] propose to find a manifold-aware gradient $x^* \in \mathcal{X}$ for a given ground truth R_{gt}, or a goal rotation denoted as R_g, where R_g is an intermediate rotation matrix between the network output R_{est} and the ground truth R_{gt}. The new gradient would then be

$$g = 2\left(x - x^*\right), \tag{4.6}$$

which is the gradient to be used to update the neural network weights.

Finding goal rotation R_g

Finding x^* is not trivial. Computing x^* involves performing a Riemannian optimization on $SO(3)$ introduced in Subsection 2.8.3 which gives

$$R_g \leftarrow R_k(-\tau \tilde{\nabla}\mathcal{L}_{(x_k)}), \tag{4.7}$$

where R_g is the goal rotation, and τ is the step size. The Riemannian gradient is along the geodesic path between R_{est} and R_{gt} on $SO(3)$. Thus R_g is noted to be an intermediate rotation matrix along the geodesic curve. As seen from Equation 4.7, R_g is dependent on the step size τ. $\tau = 0$ gives $R_g = R_{est}$, and by gradually increasing τ from 0 forces R_g along the geodesic, and making it approach R_{gt}. Figure 4.5 depicts the relation between R_{est}, R_g and R_{gt}.

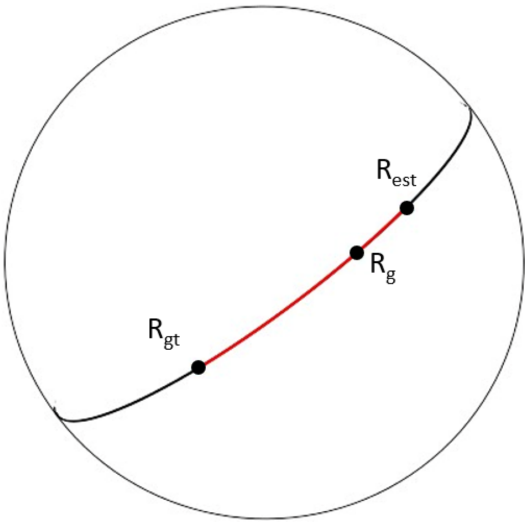


Figure 4.5.: Illustration of the relation between R_{est}, R_g and R_{gt}, where R_g is an intermediate rotation matrix on the geodesic curve between the estimation and ground truth.

Finding g_M

After computing R_g, the representation mapping ψ can be used to project from the rotation manifold $SO(3)$ onto the representation manifold $\mathcal{M}$ which gives $\hat{x}_g = \psi(R_g)$. The gradient $\hat{x}_g$ can be used to construct the manifold gradient $g_M = (x - \hat{x}_g)$, which is one of the aforementioned manifold-aware gradients.

Finding g_{PM}

Further inverting π to obtain x_g such that $\pi^{-1}(\hat{x}_g) \in \mathcal{X}$ is a non-trivial problem as there are multiple x_gs that satisfies $\pi(x_g) = \hat{x}_g$, i.e many-to-one correspondences. [1] call it a multi-ground truth problem which is due to pose symmetries and also related to the projective nature of the manifold mapping function π. Figure 4.6 illustrates various projection points $\hat{x}_{gp}$s [1].

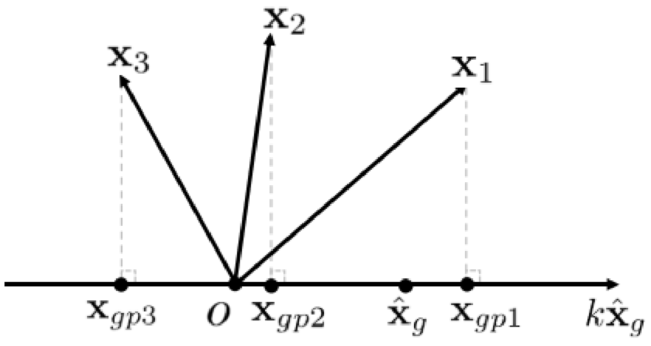


Figure 4.6.: Inversion of π to obtain x_g is a multi-ground-truth problem. Figure is from [1].

To solve this problem, [1] requires x^* to have the smallest norm to x, and opts to find the projection point x_{gp} of x to all qualified x_g given as

$$x_{gp} = \underset{\pi(x_g)=\hat{x}_g}{\arg\min} \|x - x_g\|_2 , \tag{4.8}$$

which gives $g_{PM} = (x - x_{gp})$, and is denoted as a projective manifold gradient. In [1] the RPMG-layer includes only to quaternions, 6D, 9D and 10D representations. The inverse projection with π is different for the various rotation representations. The inverse projections for the mentioned rotation representations are found in appendix A.1.

Finding g_{RPM}

The authors in [1] adds a regularization term onto g_{PM} which gives the regularized projective manifold gradient as

$$g_{RPM} = x - x_{gp} + \lambda\left(x_{gp} - \hat{x}_g\right), \tag{4.9}$$

where λ is a regularization coefficient. g_{RPM} is noted to solve a problem related to the norm of the network output, which tends to become small during training, which further will lead to convergence issues and harm to the network performance, which in [1] is denoted as a *length-vanishing* problem. It is noted in [1] that a requirement to maintain g_{RPM} is to keep λ small. In their work $\lambda = 0.01$. Note that $\lambda = 1$, g_{RPM} becomes g_M, while $\lambda = 0$, g_{RPM} gives g_{PM} [1]. Thus, hyperparameters in the algorithm of [1] are highly important for network performance. Moreover, note from Figure 4.6 that when the angle between x and $\hat{x}_g$ becomes larger than $\frac{\pi}{2}$ radians as seen for x_3, the projection x_{gp} is in the opposite direction of $\hat{x}_g$, and thus can not be mapped back to $\hat{x}_g$ by $\pi(x_{gp3}) = \hat{x}_g$, which will result in a reversed gradient [1]. To tackle this problem, the hyperparameter τ in Equation 4.7 is chosen to be small in the initial stage of training, such that R_g is close to R_{est}. During the latter stages as the network is about to converge, τ is ramped up to force R_g closer to R_{gt} for better convergence. The network is noted to be converging when the geodesic distance (Equation 2.50) between R and R_{gt} lessens.

Figure 4.7 illustrates the raw network output x mapped to $\hat{x}_g$ by π. The green arrow shows $\hat{x}_g$ of the goal rotation R_g after representation mapping $\psi(R_g)$ onto $\mathcal{M}$. The blue arrow is shown to be the inverse projection x_{gp} of $\hat{x}g$. Further adding the regularization term λ gives g_{PMG} which is shown as the purple line [1].

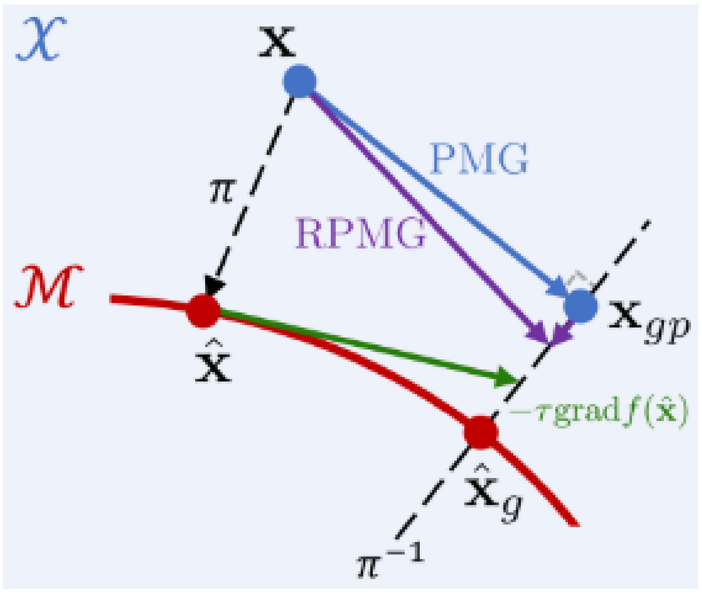


Figure 4.7.: The gradients of g_M, g_{PM} and g_{RPM} in action. Figure is from [1].

The impact of the manifold-aware g_M, g_{PM} and g_{RPM} on the quaternion, 6D, 9D and 10D representations are extensively studied in Chapter 5. The length-vanishing problem related to g_{PM} is depicted and compared against the regularized counterpart in g_{RPM}.

Chapter 5.

Objective & Simulation

5.1. Objective

5.1.1. Task

The simulation in this report is based on the contribution of Chen et al.[1], which measures the impact of the various manifold-aware gradients g_M, g_{PM} and g_{RPM} equpped on the quaternion, 6D, 9D and 10D representations. The objective in this thesis is to study the impact of the manifold-aware gradients on the generalization error produced by the various representations, where all results will be compared against each other in box-plots, error plots and tables. As mentioned in the previous chapter, the length-vanishing problem imposed by g_{PM} is depicted and compared against the regularized g_{RPM}. Noted in [1], the only requirement is to keep λ strictly larger than 0. The λ employed in [1] is set to $\lambda = 0.01$, which is also the case in this thesis along with an additional adjustment of $\lambda = 0.0005$. The new λ is then tested on the 6D, 9D and 10D representation. All simulations conducted with the RPMG-layer operates with a τ where $\tau : \tau_{initial} = \frac{1}{20} \rightarrow \tau_{final} = \frac{1}{4}$ in 10 steps as $d_{\mathcal{M}}(R_{est}, R_{gt}) \rightarrow 0$.

5.1.2. PointNet++ MSG on ModelNet40

The simulation study in Chen et al.[1] involved training and testing PointNet++ MSG on ModelNet40 [37]. ModelNet40 is a widely used benchmark for point cloud analysis. The data set consists of 12,311 CAD-generated meshes (split into 9,843 for training and 2,468 for testing) in 40 categories (such as airplane, car, guitar etc.) [37], and is a proposal from Princeton Vision & Robotics Labs to aid deep learning researchers in computer vision and robotics tasks [32].

5.1.3. Idun HPC

The simulation is conducted on Idun High-Performance Computing (Idun HPC), which uses Graphical Processing Unit (GPU) computer clusters to solve advanced computational problems [38]. Idun HPC is an initiative from the Norwegian Techincal University of Science (NTNU).

5.2. Simulation details

The simulation in this report will train, validate and test the network on meshes of various models of airplanes.The training lasts for 30k iterations and uses the Adam optimizer with the initial learning rate set to $1e^{-3}$. The learning rate is decayed by 0.7 every 3000-th iterations. A validation set of test samples is run in parallel during the training, in order to keep track of progress. Figure 5.1 shows four distinct raw points clouds of airplane models from ModelNet40 in $\mathbb{R}^{3\times5632}$.

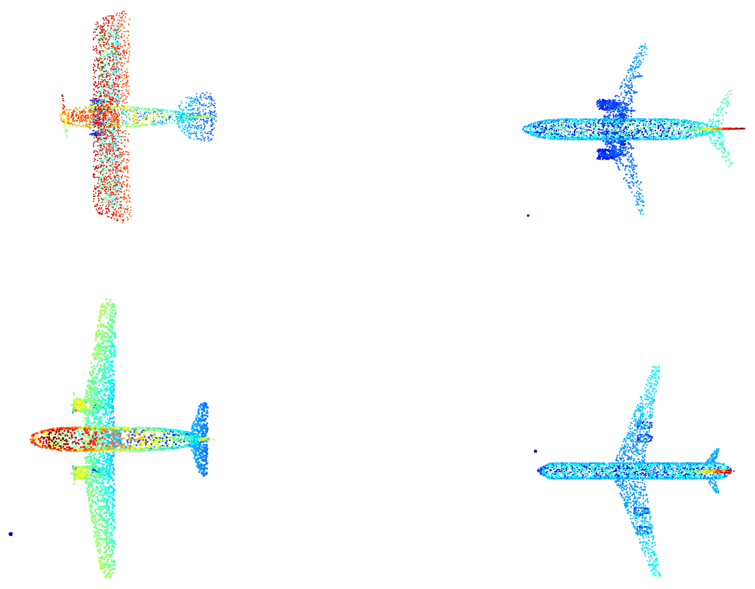


Figure 5.1.: Airplane models from ModelNet40 $\mathbb{R}^{3\times5632}$.

However, the data sets which are passed through the network for training, validation and testing are reduced to $\mathbb{R}^{3\times1024}$. This reduction is seen in Figure 5.2. The training set consists of 626 various airplane models, while the test set has 100 distinct airplane models.

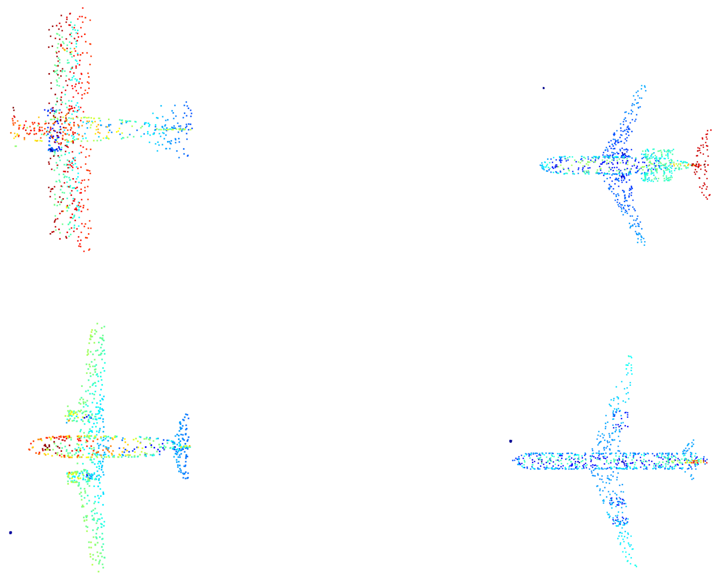


Figure 5.2.: Airplane models from ModelNet40 in $\mathbb{R}^{3\times1024}$.

The Python-function `def train_one_iteration()` is a part of loop in another Python-function called `def train(param)`. `def train_one_iteration()` accepts a training set as input, which is passed through for 30k iterations. The function takes a random batch (20 batches in this simulation) of input point clouds, and generates a batch-amount of ground-truth rotation matrices. At each iteration, the batches of training samples are passed to PointNet++ MSG in one end, and outputs batches of rotation matrices (R_{est}) in the other end. The outputted rotation matrices creates a loss with the ground-truth rotation matrices (R_{gt}). The gradient of the loss is then passed to the RPMG-layer, which leverages from Riemannian optimization to create a goal rotation R_g, which further leads to the backpropagation of the neural network weights with g_M, g_{PM} and g_{RPM}. All Python scripts for conducting the simulations are presented in Appendix B.

```python
def train_one_iteraton(pc, param, model, optimizer, iteration, tau):
    optimizer.zero_grad()
    batch=pc.shape[0]
    point_num = param.sample_num

    ###get training data######
    pc1 = torch.autograd.Variable(pc.float().cuda()) #num*3
    gt_rmat = tools.get_sampled_rotation_matrices_by_axisAngle(batch
                                        )#batch*3*3
    gt_rmats = gt_rmat.contiguous().view(batch,1,3,3).expand(batch,
                                        point_num, 3,3 ).contiguous().view
                                        (-1,3,3)
    pc2 = torch.bmm(gt_rmats, pc1.view(-1,3,1))#(batch*point_num)*3*
                                        1
    pc2 = pc2.view(batch, point_num, 3) ##batch,p_num,3

    ###network forward########
    out_rmat,out_nd = model(pc2.transpose(1,2))   #output [batch(*
                                        sample_num),3,3]

    ####compute loss##########
    if not param.use_rpmg:
        loss = ((gt_rmat - out_rmat) ** 2).mean()

    else:
        out_9d = rpmg.RPMG.apply(out_nd, tau, param.rpmg_lambda,
                                        gt_rmat, iteration)
        loss = ((gt_rmat - out_9d)**2).sum()

    loss.backward()
    optimizer.step()


    return loss
```

5.2.1. Idun HPC

Assuming the reader has access to Idun. Certain bash commands must be executed in order to conduct the simulation.

See the following command:

```
$ srun --nodes=1 --partition=GPUQ --gres=gpu:1 --time=100:00:00 --
  pty bash
$ module load PyTorch/1.7.1-fosscuda-2020b
```

5.2.2. Code compilation in Idun

After reserving the GPU-node in Subsection 5.2.1, it should be straight-forward to follow the Github repository provided by [1] to conduct the simulation. The URL of the Github repository is `https://github.com/JYChen18/RPMG.git`. When Github repository is cloned, follow the next steps to conduct a simulation:

Download dataset from ModelNet40:

```
$ cd RPMG/ModelNet_PC
$ mkdir dataset && cd dataset
$ wget https://lmb.informatik.uni-freiburg.de/resources/datasets/
  ORION/modelnet40_manually_aligned.tar
$ mkdir modelnet40 && tar xvf modelnet40_manually_aligned.tar -C
  modelnet40
$ cd ..
```

Prepocess data:

```
$ cd code
$ python prepare.py -d ../dataset/modelnet40 -c airplane
$ cd ..
```

Train and test: To train and test the network, use configuration-file in Appendix B.2.1 to set the desired properties on RPMG-layer. The instructions are given in the config-file.

```
$ cd code
$ python train.py --config example.config
$ python test.py --config example.config --rotation_map
  name_of_rot_map
$ cd ..
```

5.2.3. Transferring files to create tables and graphs

As Graphical User Interface (GUI) in Idun HPC is not available, plotting and visualizing graphs is inconvenient in Idun HPC. The train.py-file creates a folder at RPMG/ModelNet_PC/exp, which stores the weights and Tensorboard-files of the trained representation. Compiling the test.py-file stores the output in an Excel-file in RPMG/ModelNet_PC/code. Both the Excel-and Tensorboard-files were then transferred from Idun HPC to PC via WinSCP. WinSCP is a file transfer application which securely transfers files from a local computer to an external computer via a SSH protocol [39]. In the local computer, the files were used to create the graphs and tables shown in Chapter 6. The IDE used during this thesis was Spyder IDE, which is a free and open source scientific Python development environment [40].

Chapter 6.

Results & Discussion

This chapter presents the results from the simulations and a discussion of the results. The results are depicted in tables, box-plots and error plots, where the objective is to display a comparison between the various settings of rotation representations with Euclidean gradients versus the manifold-aware gradients derived by the inverse mappings of the goal rotation R_g. The results are given in geodesic errors, noted as $d_{\mathcal{M}}$-error. Table 6.1 serves an overview of all representations. The RPMG-layer is employed on quaternion, 6D, 9D and 10D representations. The length-vanishing problem imposed when using g_{PM} which returns zero gradients are also illustrated and compared against the gradients of g_{RPM} in Figure 6.10. All results are discussed in Section 6.2.

6.1. Results

6.1.1. Rotation representations

This section depicts the $d_{\mathcal{M}}$-test error of various rotation representations. The results are shown in box-plots and an error-plot. Figure 6.1 and Figure 6.2 depicts the geodesic test error. It is seen that 6D, 9D and 10D are dominant in accuracy compared to the rest, where 6D is seen to be the superior.

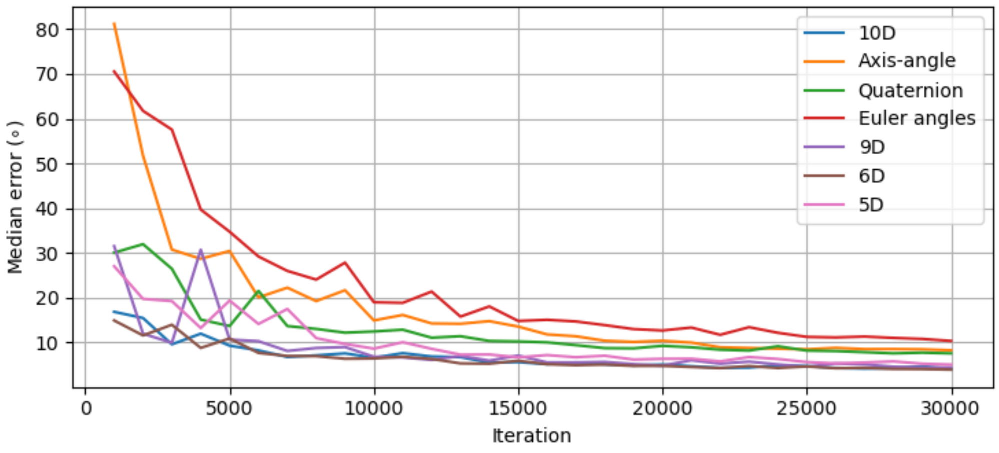


Figure 6.1.: Median $d_{\mathcal{M}}$-test error of airplane models in different iterations during training. Simulation is done without manifold-aware gradients. The plot is a replication of [6] and [8] trained on ModelNet40. 5D, 6D, 9D and 10D is shown to be the most optimal rotation representations.

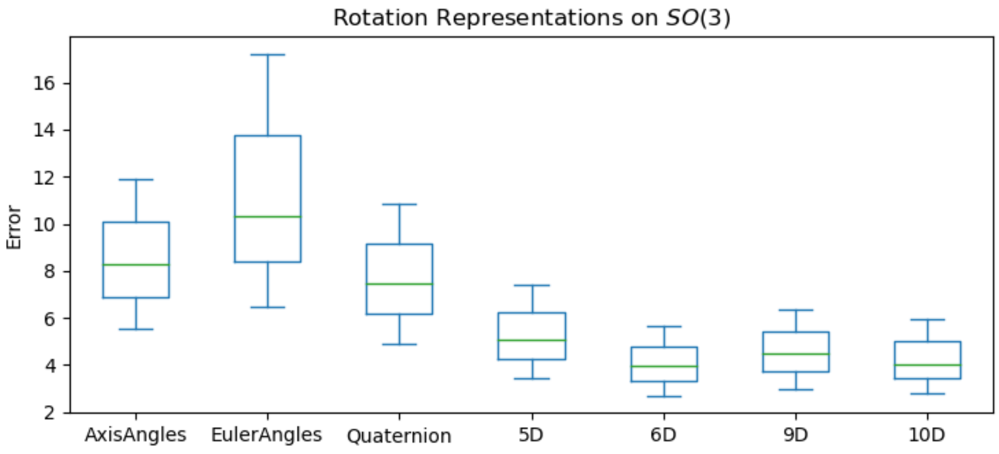


Figure 6.2.: Box plot of rotation representations without manifold-aware gradients.

6.1.2. Rotation representations with g_M

This section depicts the $d_{\mathcal{M}}$-test error of various rotation representations when using the RPMG-layer in the network. The manifold-aware gradient in this simulation is g_M, which means $\lambda = 1$. The results are shown in box-plots and an error-plot. Figure 6.3 and Figure 6.4 depicts the geodesic test error. It is seen that 6D-MG, 9D-MG and 10D-MG are dominant in accuracy compared to the Quaternion-MG.

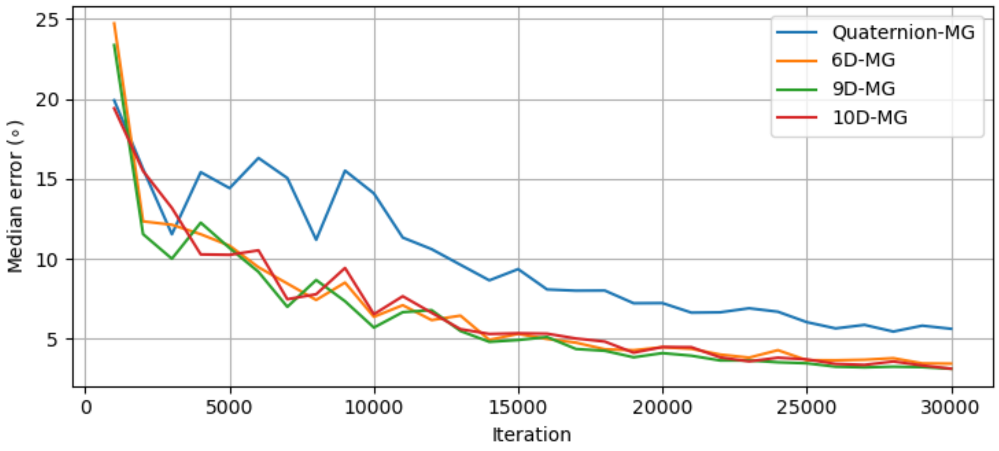


Figure 6.3.: Median $d_{\mathcal{M}}$-test error of airplane models in different iterations during training. Simulation is performed on an RPMG-layer using g_M as the manifold-aware gradient.

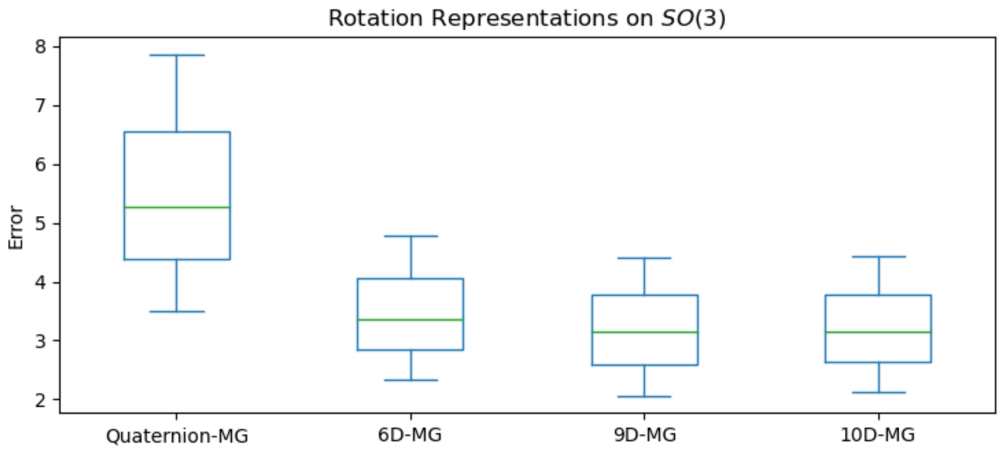


Figure 6.4.: Box plot of rotation representations using g_M as a manifold-aware gradient.

6.1.3. Rotation representations with g_{PM}

This section depicts the $d_{\mathcal{M}}$-test error of various rotation representations when using the RPMG-layer in the network. The manifold-aware gradient in this simulation is g_{PM}, which means $\lambda = 0$. The results are shown in box-plots and an error-plot. Figure 6.5 and Figure 6.6 depicts the geodesic test error. It is obvious that the results using g_{PM} are not sufficient, as none of the representations converges.

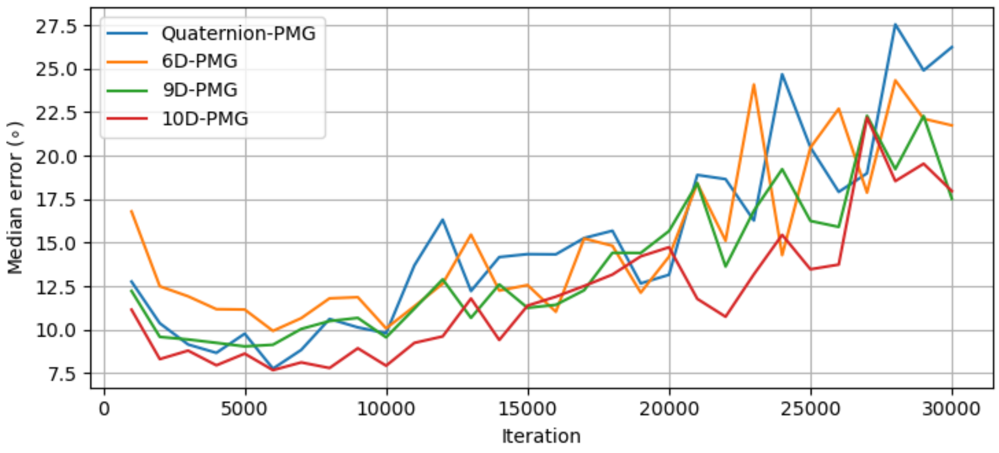


Figure 6.5.: Median $d_{\mathcal{M}}$-test error of airplane models in different iterations during training. Simulation is performed on an RPMG-layer using g_{PM} as the manifold-aware gradient.

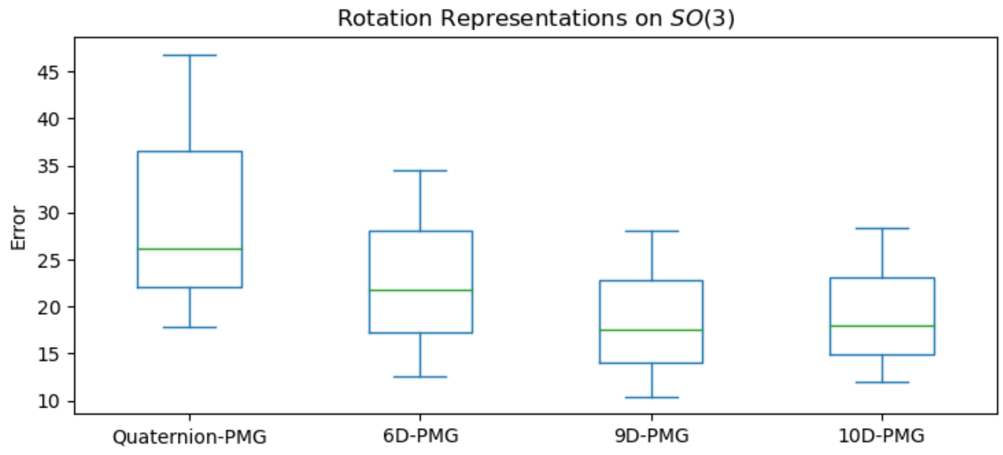


Figure 6.6.: Box plot of rotation representations using g_{PM} as a manifold-aware gradient.

6.1.4. Rotation representations with g_{RPM}

This section depicts the $d_{\mathcal{M}}$-test error of various rotation representations when using the RPMG-layer in the network. The manifold-aware gradient in this simulation is g_{RPM}, which in this simulation uses $\lambda = 0.01$. The results are shown in box-plots and an error-plot. Figure 6.7 and Figure 6.8 depicts the geodesic test error. It is seen that 6D-RPMG, 9D-RPMG and 10D-RPMG are superior of Quaternion-MG.

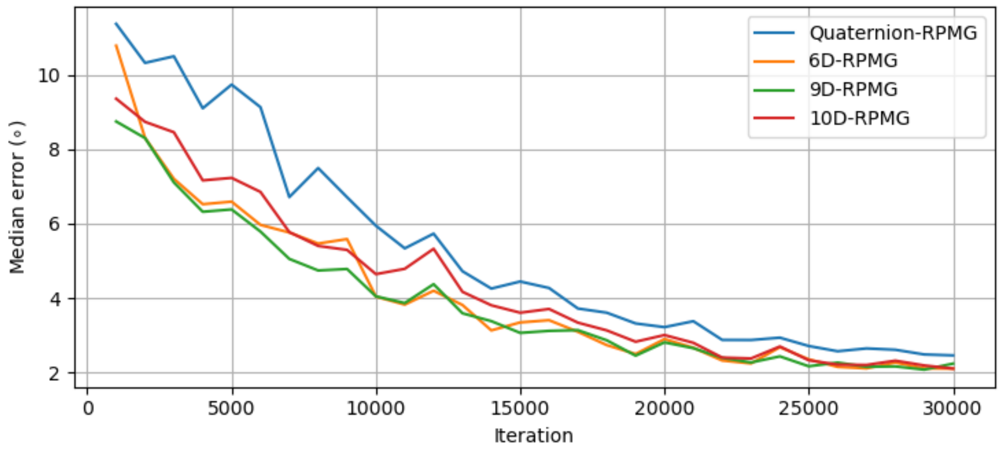


Figure 6.7.: Median $d_{\mathcal{M}}$-test error of airplane models in different iterations during training. Simulation is performed on an RPMG-layer using g_{RPM} as the manifold-aware gradient.

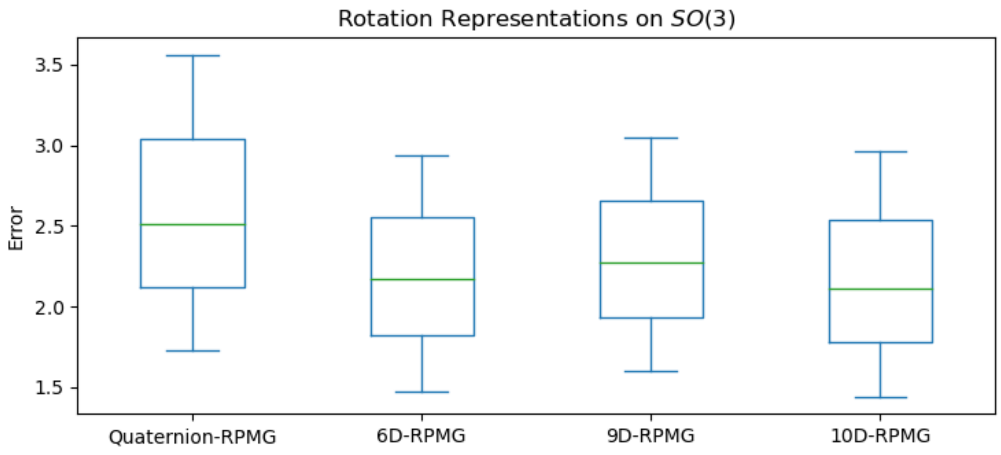


Figure 6.8.: Box plot of rotation representations using g_{RPM} as a manifold-aware gradient.

6.1.5. Length-vanishing problem

The length-vanishing problem when using g_{PM} is depicted in Figure 6.9, and compared against the gradient found from g_{RPM}, which is illustrated in Figure 6.10. The representations simulated in this specific simulation are based on the parameterization from 6D, 9D and 10D network output. The plot shows the relation between the gradients and ℓ_2-norm of the gradients at a given iteration during training.

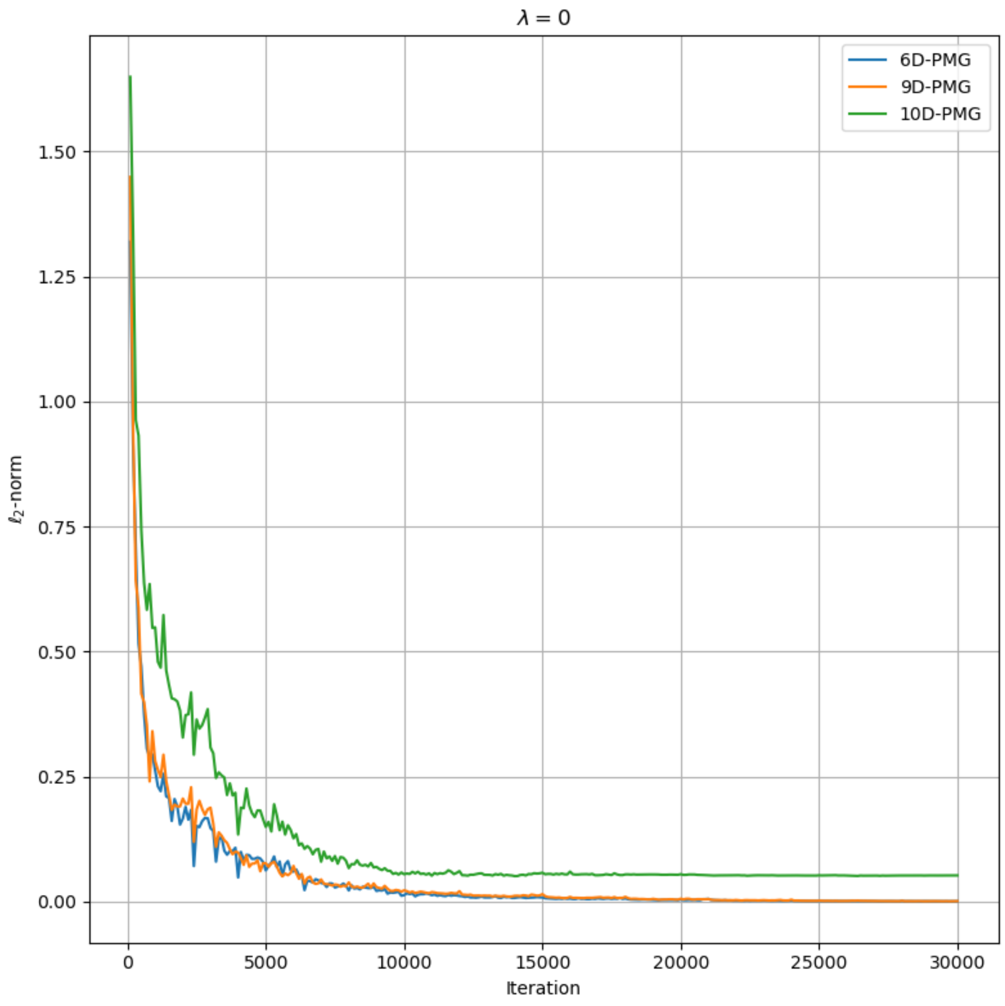


Figure 6.9.: Using g_{PM} imposes vanishing gradients.

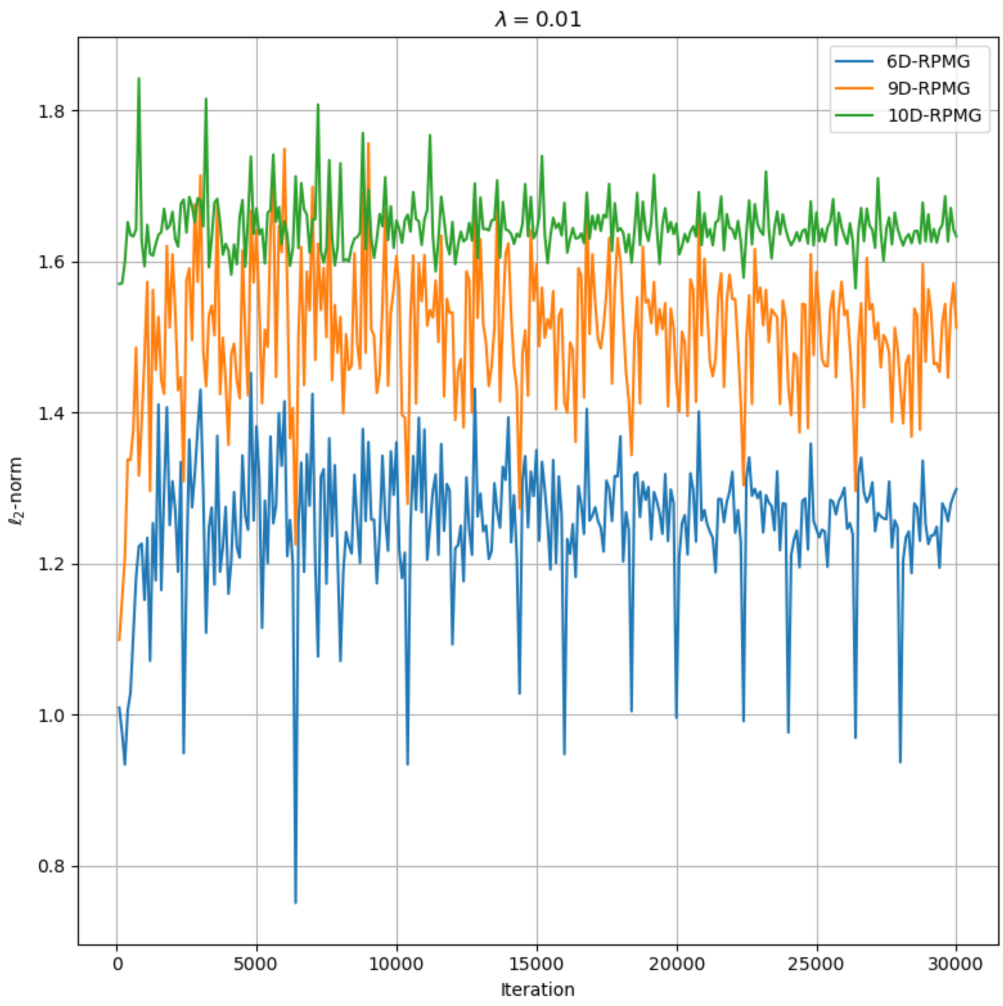


Figure 6.10.: The g_{RPM} gradient has stable gradients in comparison to g_{PM}.

6.1.6. Overview of results

This section depicts the comparison when employing/not employing the RPMG-layer on various representations. The results are given in Figure 6.11 and Table 6.1.

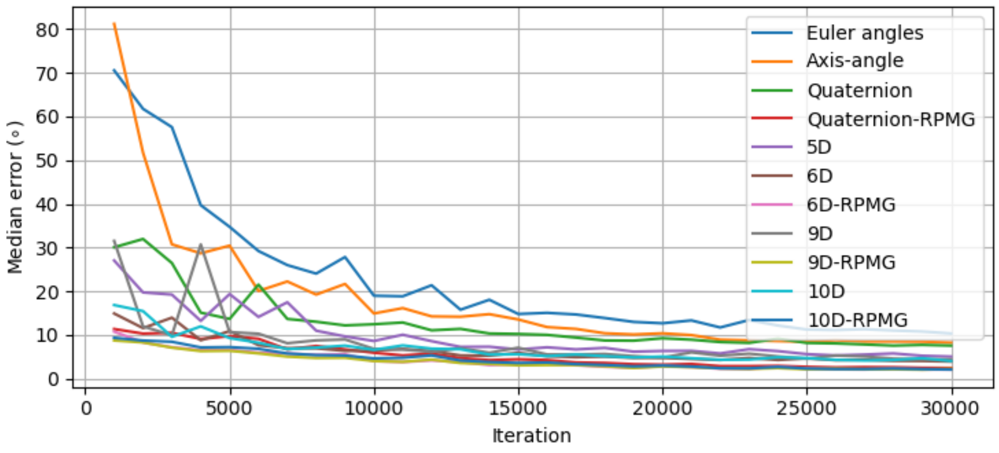


Figure 6.11.: Error plot of rotation representations with and without the RPMG-layer.

The graph shows the overall performance of all the tested rotation representation in the simulations. It is seen from Table 6.1 that 6D-RPMG enjoys the most optimal performance compared to the rest, with a 5° geodesic accuracy of 94.9 %.

Rotation Representation	**Loss**	**Min**	**Md**	**5° Acc**
Axis-angle	1.361	0.72	8.27	0.198
Euler angles	2.877	0.63	10.35	0.134
Quaternion	1.087	0.37	7.45	0.264
5D	0.356	0.23	5.06	0.493
6D	0.197	0.37	3.95	**0.68**
9D	0.304	0.25	4.51	0.576
10D	0.228	0.58	4.05	0.632
Quaternion-MG	0.478	0.31	5.26	0.469
6D-MG	0.136	0.16	3.37	0.771
9D-MG	0.127	0.14	3.14	0.811
10D-MG	0.147	0.34	3.14	**0.813**
Quaternion-PMG	10.805	5.04	26.24	0.0
6D-PMG	7.978	2.09	21.74	0.05
9D-PMG	5.872	0.19	17.53	0.045
10D-PMG	5.042	1.44	17.97	0.027
Quaternion-RPMG	0.151	0.3	2.51	0.899
6D-RPMG	0.066	0.24	2.17	**0.949**
9D-RPMG	0.076	0.28	2.27	0.946
10D-RPMG	0.074	0.14	2.11	0.943

Table 6.1.: A comparison of rotation representations by loss, minimum- and median $d_{\mathcal{M}}$-test error, along with 5° accuracy of $d_{\mathcal{M}}$-test errors after 30k training steps. Min, Md and Acc are abbreviations of minimum, median and 5° accuracy. The most optimal 5° Acc is marked in **blue**, and belongs to 6D-RPMG, while **red** colorization specifies the superior representation within its respective domain.

Rotation Representation	**Loss**	**Min**	**Md**	**5° Acc**
6D-RPMG	0.489	0.21	2.481	0.871
9D-RPMG	0.411	0.17	2.362	**0.896**
10D-RPMG	0.556	0.16	2.782	0.803

Table 6.2.: This table shows the results on 6D, 9D and 10D representations when $\lambda = 0.0005$. The results proves that the only requirement is to keep $\lambda > 0$ to maintain great generalization errors.

6.2. Discussion

The obtained results from the simulations proves that the contribution of Chen et al.[1] optimizes the previous work of Zhou et al.[6], Levinson et al.[7], Peretroukhin et al.[19] and Brègier [8]. Table 6.1 indicates that all representations using the RPMG-layer with manifold-aware gradients enjoys superior performance over the regular rotation representations using Euclidean gradients. The 6D-RPMG representation is overall the best representation, along with a tight follow-up from 9D-RPMG and 10D-RPMG. It is also seen that the g_{PM}-gradient is quite unacceptable, and from Figure 6.5, it is seen that the error plot does not converge for any representation, which is due to the length-vanishing problem depicted in Subsection 6.1.5, where it is that the ℓ_2-norm of the gradient of g_{PM} converges to 0. This challenge is tackled by adding the regularization term in Figure 4.7, which gives stable gradients depicted in Figure 6.10. It is noted in [1] that the only requirement is setting $\lambda > 0$. This claim is tested in Table 6.2 as $\lambda = 0.0005$. As seen from the results, the statement proves to be correct. With a λ close to 0, 9D-RPMG shows to be the better representation.

Chapter 7.

Conclusion

This master's thesis has studied learning-friendly rotation representations in deep rotation regression when using PointNet++ MSG as the backbone neural network. It has been studied that learning-friendly rotation representations are strongly related to topological concepts on homeomorphism between smooth manifolds. The homeomorphism preserves properties during bijective mappings between manifolds. The manifold mapping of interest in this study are between the Euclidean $\mathbb{R}^n$ and the rotation space $SO(3)$. It is seen that when the full rotation space ($\theta = [0, 2\pi]$) is required, certain n-dimensional neural network outputs in $\mathbb{R}^n$ are discontinuous, and imposes difficulties when training on a continuous neural network. The proposal of Zhou et al.[6] along with the contributions of Romain Brègier [8], Levinson et al.[7] and Peretroukhin et al.[19] proves that discontinuous neural network outputs exists only for vectors less than 5 dimensions. The simulation in this thesis has proved that 5D, 6D, 9D and 10D representations are better suited for neural network learning in deep rotation regression, as those rotation representations are homeomorphic to $SO(3)$. Furthermore, an additional study from Chen et al.[1] studies the application of geometric deep learning on various rotation representations. It is seen from the simulations that by employing Riemannian optimization to derive manifold-aware gradients through a goal rotation R_g, consistently improves generalization on quaternion, 6D, 9D and 10D representations when using g_M and g_{RPM} as the manifold-aware gradients.

References

[1] J. Chen, Y. Yin, T. Birdal, B. Chen, L. J. Guibas, and H. Wang, "Projective Manifold Gradient Layer for Deep Rotation Regression," *CoRR*, vol. abs/2110.11657, 2021. arXiv: 2110.11657. [Online]. Available: https://arxiv.org/abs/2110.11657.

[2] X. Huang, G. Mei, J. Zhang, and R. Abbas, "A Comprehensive Survey on Point Cloud Registration," *CoRR*, vol. abs/2103.02690, 2021. arXiv: 2103.02690. [Online]. Available: https://arxiv.org/abs/2103.02690.

[3] H. Yang, J. Shi, and L. Carlone, "TEASER: Fast and Certifiable Point Cloud Registration," *CoRR*, vol. abs/2001.07715, 2020. arXiv: 2001.07715. [Online]. Available: https://arxiv.org/abs/2001.07715.

[4] Q. Zhou, J. Park, and V. Koltun, "Fast Global Registration," in *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., ser. Lecture Notes in Computer Science, vol. 9906, Springer, 2016, pp. 766–782. DOI: 10.1007/978-3-319-46475-6_47. [Online]. Available: https://doi.org/10.1007/978-3-319-46475-6%5C_47.

[5] G. Gao, M. Lauri, J. Zhang, and S. Frintrop, "Occlusion Resistant Object Rotation Regression from Point Cloud Segments," 2018. DOI: 10.48550/ARXIV.1808.05498. [Online]. Available: https://arxiv.org/abs/1808.05498.

[6] Y. Zhou, C. Barnes, J. Lu, J. Yang, and H. Li, "On the Continuity of Rotation Representations in Neural Networks," 2018. DOI: 10.48550/ARXIV.1812.07035. [Online]. Available: https://arxiv.org/abs/1812.07035.

[7] J. Levinson, C. Esteves, K. Chen, N. Snavely, A. Kanazawa, A. Rostamizadeh, and A. Makadia, "An Analysis of SVD for Deep Rotation Estimation," 2020. DOI: 10.48550/ARXIV.2006.14616. [Online]. Available: https://arxiv.org/abs/2006.14616.

[8] R. Brégier, "Deep Regression on Manifolds: A 3D Rotation Case Study," 2021. DOI: 10.48550/ARXIV.2103.16317. [Online]. Available: https://arxiv.org/abs/2103.16317.

[9] K. B. Petersen and M. S. Pedersen, *The Matrix Cookbook*, 2008. [Online]. Available: http://matrixcookbook.com/.

[10] A. Barrau, "Non-Linear State Error Based Extended Kalman Filters with Applications to Navigation.," 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01344622.

[11] A. Iserles, H. Z. Munthe-Kaas, S. P. Nørsett, and A. Zanna, "Lie-group Methods," *Acta Numerica*, 2000. [Online]. Available: http://hans.munthe-kaas.no/work/Blog/Entries/2000/1/1_Article__Lie-group_methods_files/iserles00lgm.pdf.

[12] A. Sjoberg and O. Egeland, "An EKF for Lie Groups with Application to Crane Load Dynamics," *Modeling, Identification and Control*, 2019, ISSN: 1890–1328.

[13] S. Ovchinnikov, *Functional Analysis*. Springer, 2018. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-319-91512-8.pdf.

[14] A. Matsumoto and F. Szidarovszky, *Game Theory and Its Applications*, 1st ed. 2016. Springer, 2016. [Online]. Available: https://EconPapers.repec.org/RePEc:spr:sprbok:978-4-431-54786-0.

[15] M. Moakher, "Means and Averaging in the Group of Rotations," *SIAM Journal on Matrix Analysis and Applications*, vol. 24, no. 1, 2002. DOI: 10.1137/S0895479801383877. [Online]. Available: https://doi.org/10.1137/S0895479801383877.

[16] Zhang Yanchun and Xu, Guandong, and Ozsu, M. Tamer, *Singular Value Decomposition*. Boston, MA: Springer US, 2009, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_538. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_538.

[17] I. Yanovsky, *QR Decomposition with Gram-Schmidt*. [Online]. Available: https://www.math.ucla.edu/~yanovsky/Teaching/Math151B/handouts/GramSchmidt.pdf.

[18] K. M. Lynch and F. C. Park, *Modern Robotics: Mechanics, Planning, and Control*, 1st. Cambridge University Press, 2017, ISBN: 1107156300.

[19] V. Peretroukhin, M. Giamou, W. N. Greene, D. Rosen, J. Kelly, and N. Roy, "A Smooth Representation of Belief over SO(3) for Deep Rotation Learning with Uncertainty," in *Robotics: Science and Systems XVI*, Robotics: Science and Systems Foundation, Jul. 2020. DOI: 10.15607/rss.2020.xvi.007. [Online]. Available: https://doi.org/10.15607%2Frss.2020.xvi.007.

[20] S. C. Carlson, *Topology*, 2017. [Online]. Available: https://www.britannica.com/science/topology.

[21] G. E.Bredon, *Topology and Geometry*. Springer New York, 1993. DOI: https://doi.org/10.1007/978-1-4757-6848-0.

[22] D. Chakraborty and K. Sarma, "A Study on Blind Source Separation using ICA Algorithm in Terms of Invertible System," *Journal of Basic and Applied Engineering Research*, 2019, ISSN: 2350-0077.

[23] A. Kuronya, *Introduction to Topology*. CreateSpace Independent Publishing Platform, 2014, ISBN: 9781502795939. [Online]. Available: https://books.google.no/books?id=3Ov0oQEACAAJ.

[24] A. Sagle and R. Walde, *Introduction to Lie Groups and Lie Algebras*, ser. Pure and Applied Mathematics; A Series of Monographs and Tex. Academic Press, 1973, ISBN: 9780126145502. [Online]. Available: https://books.google.no/books?id=3TOmnQEACAAJ.

[25] S. Lovett, *Differential Geometry of Manifolds*, ser. Textbooks in mathematics. CRC Press, 2019, ISBN: 9780367180461. [Online]. Available: https://books.google.no/books?id=vB1AxwEACAAJ.

[26] J. Wilson, *Manifolds*, 2012. [Online]. Available: http://www.math.lsa.umich.edu/~jchw/WOMPtalk-Manifolds.pdf.

[27] Michaelmas, *Smooth Manifolds and Tangent space: Outline*, 2013. [Online]. Available: https://www.maths.dur.ac.uk/users/anna.felikson/RG/RG13/outline1.pdf.

[28] John M. Lee, *Introduction to Smooth Manifolds*. Springer, 2000. DOI: 10.1007/978-1-4419-9982-5.

[29] F. C. Park, "Distance Metrics on the Rigid-Body Motions with Applications to Mechanism Design," *Journal of Mechanical Design*, vol. 117, no. 1, pp. 48–54, 1995, ISSN: 1050-0472. DOI: 10.1115/1.2826116. [Online]. Available: https://doi.org/10.1115/1.2826116.

[30] R. I. Hartley, J. Trumpf, Y. Dai, and H. Li, "Rotation Averaging," *International Journal of Computer Vision*, vol. 103, pp. 267–305, 2012.

[31] A. de Ruiter and J. Forbes. [Online]. Available: https://sdac.blog.ryerson.ca/files/2016/02/WahbaSOnpaperJASrevision4.pdf.

[32] S. A. Bello, S. Yu, and C. Wang, "Review: Deep learning on 3d point clouds," *CoRR*, vol. abs/2001.06280, 2020. arXiv: 2001.06280. [Online]. Available: https://arxiv.org/abs/2001.06280.

[33] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," 2016. DOI: 10.48550/ARXIV.1612.00593. [Online]. Available: https://arxiv.org/abs/1612.00593.

[34] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, “PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space,” 2017. DOI: 10.48550/ARXIV.1706.02413. [Online]. Available: https://arxiv.org/abs/1706.02413.

[35] W. J. Marais, R. E. Holz, J. S. Reid, and R. M. Willett, “Leveraging Spatial Textures, Through Machine Learning, to Identify Aerosols and Distinct Cloud Types from Multispectral Observations,” *Atmospheric Measurement Techniques*, vol. 13, no. 10, pp. 5459–5480, 2020. DOI: 10.5194/amt-13-5459-2020. [Online]. Available: https://amt.copernicus.org/articles/13/5459/2020/.

[36] S. Xiang and H. Li, “Revisiting the Continuity of Rotation Representations in Neural Networks,” 2020. DOI: 10.48550/ARXIV.2006.06234. [Online]. Available: https://arxiv.org/abs/2006.06234.

[37] *Princeton ModelNet*. [Online]. Available: https://modelnet.cs.princeton.edu/.

[38] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, “EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure,” 2019. arXiv: 1912.05848 [cs.DC].

[39] [Online]. Available: https://winscp.net/eng/index.php.

[40] [Online]. Available: https://www.spyder-ide.org/.

Appendix A.

Mathematical Formulations

Mathematical formulations which were too extensive to include in the main sections of the thesis are included in this appendix. In section A.1 the inverse image projection $x_g = \pi^{-1}(\hat{x}_g)$ quaternion, 6D, 9D and 10D representations are presented.

A.1. Derivation of inverse projections

All of the following derivations are from Chen et al. [1], and is used to compute x_g in g_{PM} by using the next goal $\hat{x}_g$ in an inversion step by π^{-1} from the representation manifold $\mathcal{M}$ to the ambient space $\mathcal{X}$. The following codes are directly extracted from [1].

A.1.1. Quaternion

$$x_{gp} = \underset{x_g \in \pi_q^{-1}(\hat{x}_g)}{\arg\min} \ \|x_g - x\|_2^2 , \tag{A.1}$$

where x is the raw output of our network in ambient space $\mathbb{R}^4$, $\hat{x}_g$ is the next goal in representation manifold S^3, and x_g is the variable to optimize in ambient space $\mathbb{R}^4$. Recall $\pi_q^{-1}(\hat{x}_g) = \{x \mid x = k\hat{x}_g, k \in \mathbb{R} \text{ and } k > 0\}$, and

$$\|x - x_g\|_2^2 = x^2 - 2kx \cdot \hat{x}_g + k^2\hat{x}_g^2 \tag{A.2}$$

Without considering the condition of $k > 0$, it is noted when $k = \frac{x \cdot \hat{x}_g}{\hat{x}_g^2} = x \cdot \hat{x}_g$ the target formula reaches minimum. Note that when using a small τ, the angle between $\hat{x}_g$ and x is always very small, which means the condition of $k = x \cdot \hat{x}_g > 0$

can be satisfied naturally. For the sake of simplicity and consistency of gradient, the limitation of k is ignored no matter what value τ takes. Therefore, the inverse projection is $x_{gp} = (x \cdot \hat{x}_g)\,\hat{x}_g$.

A.1.2. 6D representation

For 6D representations, the following must be solved

$$[u_{gp}, v_{gp}] = \underset{[u_g, v_g] \in \pi_{6D}^{-1}([\tilde{u}_g, \tilde{v}_g])}{\arg\min} \left(\|u_g - u\|_2^2 + \|v_g - v\|_2^2 \right) \tag{A.3}$$

where $[u, v]$ is the raw output of network in ambient space $\mathbb{R}^6$, $[\hat{u}_g, \hat{v}_g]$ is the next goal in representation manifold $\mathcal{V}_2\left(\mathbb{R}^3\right)$ and $[u_g, v_g]$ is the variable to optimize in ambient space $\mathbb{R}^6$. Recall $\pi_{6D}^{-1}\left([\hat{u}_g, \hat{v}_g]\right) = \{[k_1\hat{u}_g, k_2\hat{u}_g + \ k_3\hat{v}_g] \mid k_1, k_2, k_3 \in \mathbb{R}$ and $k_1, k_3 > 0\}$. It is seen that u_g and v_g are independent, and u_g is similar to the situation of quaternion. So the only considered part is v_g given below

$$\|v - v_g\|_2^2 = v^2 + k_2^2\hat{u}_g^2 + k_3^2\hat{v}_g^2 - 2k_2 v \cdot \hat{u}_g - 2k_3 v \cdot \hat{v}_g \tag{A.4}$$

For the similar reason as quaternion, the condition of $k_3 > 0$ is ignored and it is seen when $k_2 = v \cdot \hat{u}_g$ and $k_3 = v \cdot \hat{v}_g$, the target formula reaches minimum. Therefore, the inverse projection is $[u_{gp}, v_{gp}] = [(u \cdot \hat{u}_g)\,\hat{u}_g, (v.\ \ \hat{u}_g)\,\hat{u}_g + (v \cdot \hat{v}_g)\,\hat{v}_g]$.

A.1.3. 9D representation

For the 9D representation, obtaining the inverse image π_{9D}^{-1} is not so obvious. Recall $\pi_{9D}(x) = U\Sigma' V^\top$, where U and V are left and right singular vectors of x decomposed by SVD expressed as $x = U\Sigma V^\top$, and $\Sigma' = \operatorname{diag}\left(1, 1, \det\left(UV^\top\right)\right)$.

Lemma A.1.1 *The inverse image* $\pi_{9D}^{-1}(R_g) = \left\{SR_g \mid S = \ S^\top\right\}$ *satisfies that* $\{x_g \mid \pi_{9D}(x_g) = R_g\} \subset \pi_{9D}^{-1}(R_g)$.

Proof:. To find a suitable π_{9D}^{-1}, the most straightforward way is to only change the singular values $\Sigma_g = \operatorname{diag}(\lambda_0, \lambda_1, \lambda_2)$, where $\lambda_0, \lambda_1, \lambda_2$ can be arbitrary scalars, and recompose the $x_g = U\Sigma_g V^\top$.

However, it is argued that this simple method will fail to capture the entire set of $\{x_g \mid \pi_{9D}(x_g) = R_g\}$, because different U' and V' can yield the same rotation R_g. In fact, U_g can be arbitrary if $x_g = U_g\Sigma_g V_g^\top$ and $U_g\Sigma_g' V_g^\top = R_g$.

Assuming R_g is known, one can replace $V_g^\top$ by R_g and express x_g in a different way: $x_g = U_g \Sigma_g \frac{1}{\Sigma_g'} U_g^{-1} R_g$. Notice that $U_g \Sigma_g \frac{1}{\Sigma_g'} U_g^{-1}$ must be a symmetry matrix since U_g is an orthogonal matrix. Therefore, $\{x_g \mid \pi_{9D}(x_g) = R_g\} \subseteq \pi_{9D}^{-1}(R_g) = \left\{SR_g \mid S = S^\top\right\}$.

Note that such $x_g \in \pi_{9D}^{-1}(R_g)$ can't ensure $\pi_{9D}(x_g) = R_g$, because in the implementation of SVD, the order and the sign of three singular values are constrained, which is not taken into consideration. Therefore, $\{x_g \mid \pi_{9D}(x_g) = R_g\} \neq \pi_{9D}^{-1}(R_g)$. Then one must solve

$$x_{gp} = \underset{x_g \in \pi_{9D}^{-1}(R_g)}{\arg\min} \; \|x_g - x\|_2^2 \tag{A.5}$$

where x is the raw output of our network in ambient space $\mathbb{R}^{3\times3}$, $\hat{x}_g$ is the next goal in representation manifold SO(3), and x_g is the variable to optimize in ambient space $\mathbb{R}^{3\times3}$. One can further transform the objective function as below:

$$\|x_g - x\|_2^2 = \|SR_g - x\|_2^2 = \left\|S - xR_g^\top\right\|_2^2 \tag{A.6}$$

Now one can easily find when S equals to the symmetry part of $xR_g^\top$, the target formula reaches minimum. Therefore, the inverse projection admits a simple form

$$x_{gp} = \frac{xR_g^\top + R_g x^\top}{2} R_g \tag{A.7}$$

A.1.4. 10D representation

10D representation Recall the manifold mapping π_{10D} :

$$\mathbb{R}^{10} \to \mathcal{S}^3, \pi_{10D}(x) = \min_{q \in \mathcal{S}^3} q^\top A(x) q, \text{ in which} \tag{A.8}$$

$$A(\theta) = \begin{pmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \\ \theta_2 & \theta_5 & \theta_6 & \theta_7 \\ \theta_3 & \theta_6 & \theta_8 & \theta_9 \\ \theta_4 & \theta_7 & \theta_9 & \theta_{10} \end{pmatrix}. \tag{A.9}$$

One must solve

$$x_{gp} = \underset{A(x_g)q_g=\lambda q_g}{\arg\min} \ \|x_g - x\|_2^2 \, , \tag{A.10}$$

where x is the raw output of our network in ambient space $\mathbb{R}^{10}$, q$_g$ is the next goal in representation manifold $\mathcal{S}^3$, and x_g is the variable to optimize in ambient space $\mathbb{R}^{10}$. Note that λ is also a variable to optimize. For the similar reason as before, for the sake of simplicity and consistency of analytical solution, here one also need to relax the constraint that λ should be the smallest eigenvalue of $A\left(x_g\right)$.

To solve Equation A.9, $A\left(x_g\right) q_g = \lambda q_g$ is rewritten as

$$M\Delta x = \lambda q_g - A(x)q_g \tag{A.11}$$

where $\Delta x = x_g - x$ and

$$M = \begin{pmatrix} q_1 & q_2 & q_3 & q_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & q_1 & 0 & 0 & q_2 & q_3 & q_4 & 0 & 0 & 0 \\ 0 & 0 & q_1 & 0 & 0 & q_2 & 0 & q_3 & q_4 & 0 \\ 0 & 0 & 0 & q_1 & 0 & 0 & q_2 & 0 & q_3 & q_4 \end{pmatrix} \tag{A.12}$$

where $q_g = (q_1, q_2, q_3, q_4)^\top$. For simplicity, we denote $\lambda q_g - A(x)q_g$ as b.

Once one have finished the above steps for preparation, λ and Δx must be solved for the minimal problem by two steps as below. First, one assumes λ is known and the problem becomes that given M and b, we need to find the best Δx to minimize $\|\Delta x\|_2^2$ with the constraint $M\Delta x = b$. This is a typical quadratic optimization problem with linear equality constraints, and the analytical solution satisfies

$$\begin{pmatrix} I & M^\top \\ M & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \tag{A.13}$$

where v is a set of Lagrange multipliers which come out of the solution alongside Δx, and $\begin{pmatrix} I & M^\top \\ M & 0 \end{pmatrix}$ is called KKT matrix. Since this matrix has full rank almost everywhere, we can multiple the inverse of this KKT matrix in both sides of Equation A.13 and lead to the solution of Δx as below:

$$\begin{pmatrix} \Delta x \\ v \end{pmatrix} = \begin{pmatrix} I & M^\top \\ M & 0 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ b \end{pmatrix} \tag{A.14}$$

Recall that $b = \lambda q_g - A(x) q_g$, therefore until now one had the solution of Δx with respect to each λ :

$$\Delta x = \begin{pmatrix} \Delta x \\ v \end{pmatrix}_{0:10} = K\left(\lambda q_g - A(x) q_g\right) = \lambda S - T \tag{A.15}$$

in which K is the upper right part of the inverse of the KKT matrix $K = \left[\begin{pmatrix} I & M^\top \\ M & 0 \end{pmatrix}^{-1}\right]_{10:14,0:10}$, $S = K q_g$ and $T = K A(x) q_g$

Next, one must optimize λ to minimize the objective function $\|\Delta x\|_2^2$. In fact, using the results of Equation A.15, $\|\Delta x\|_2^2$ becomes a quadratic functions on λ, thus one can simply <get the final analytical solution of λ and x_{gp} :

$$\begin{cases} \lambda = \frac{\left(S^\top T + T^\top S\right)}{2 S^\top S} \\ x_{gp} = x + \lambda S - T \end{cases} \tag{A.16}$$

Appendix B.

Code Listing

This chapter serves the Python scripts from the Github-repository of Chen et al. [1] used to conduct the simulation in chapter 5. The Github-files are from folders inside a Github-repository. To show which folder a given script belongs to, the folder will be denoted as for example RPMG/ModelNet_PC/code, which means the folder code in the ModelNet_PC-folder in RPMG. Codes that were not used are not given in the appendix.

B.1. RPMG/ModelNet_PC/code/

B.1.1. config.py

```python
import tensorboardX
from os.path import join as pjoin
import configparser

class Parameters():
    def __init__(self):
        super(Parameters, self).__init__()


    def read_config(self, fn):
        config = configparser.ConfigParser()
        config.read(fn)
        self.exp_folder= config.get("Record","exp_folder")
        self.data_folder=config.get("Record", "data_folder")
        self.write_weight_folder= pjoin(self.exp_folder, 'weight')
        logdir= pjoin(self.exp_folder, 'log')
        self.logger = tensorboardX.SummaryWriter(logdir)

        self.lr =float( config.get("Params", "lr"))
        self.start_iteration=int(config.get("Params","
                                          start_iteration"))
        self.total_iteration=int( config.get("Params", "
                                    total_iteration"))
        self.save_weight_iteration=int( config.get("Params", "
                                    save_weight_iteration"))

        self.out_rotation_mode = config.get("Params","
                                    out_rotation_mode")

        self.use_rpmg = bool(int(config.get("Params", "use_rpmg")))
        self.rpmg_tau_strategy = int(config.get("Params", "
                                    rpmg_tau_strategy"))
        self.rpmg_lambda = float(config.get("Params", "rpmg_lambda")
                                    )
        self.sample_num = int(config.get("Params", "sample_num"))
        self.device = int(config.get("Params","device"))
        self.batch = int (config.get("Params","batch"))
```

B.1.2. dataset.py

```

import torch
import os
import numpy as np

class ModelNetDataset(torch.utils.data.Dataset):
    def __init__(self, data_folder,sample_num=1024):
        super(ModelNetDataset, self).__init__()
        self.paths = [os.path.join(data_folder, i) for i in os.
                                    listdir(data_folder)]
        self.sample_num = sample_num
        self.size = len(self.paths)
        print(f"dataset size: {self.size}")

    def __getitem__(self, index):
        fpath = self.paths[index % self.size]
        pc = np.loadtxt(fpath)
        pc = np.random.permutation(pc)
        return pc[:self.sample_num, :].astype(float)

    def __len__(self):
        return self.size
```

B.1.3. prepare.py

```
'''
from mesh to normalized pc
'''
import numpy as np
import torch
import os
from os.path import join as pjoin
import trimesh
import argparse
import sys
import tqdm
BASEPATH = os.path.dirname(__file__)
sys.path.insert(0,pjoin(BASEPATH, '../..'))
import utils.tools as tools

def pc_normalize(pc):
    centroid = (np.max(pc, axis=0) + np.min(pc, axis=0)) /2
    pc = pc - centroid
    scale = np.linalg.norm(np.max(pc, axis=0) - np.min(pc, axis=0))
    pc = pc / scale
    return pc, centroid, scale

if __name__ == "__main__":
    arg_parser = argparse.ArgumentParser()
    arg_parser.add_argument("-d", "--data_dir", type=str, default='
                                    dataset/
                                    modelnet40_manually_aligned', help
                                    ="Path to modelnet dataset")
    arg_parser.add_argument("-c", "--category", type=str, default='
                                    airplane', help="category")
    arg_parser.add_argument("-f", "--fix_test", action='store_false'
                                    , help="for fair comparision")
    args = arg_parser.parse_args()

    sample_num = 4096
    for mode in ['train', 'test']:
        in_folder = pjoin(args.data_dir, args.category, mode)
        out_folder = pjoin(args.data_dir, args.category, mode + '_pc
                                    ')
        os.makedirs(out_folder, exist_ok=True)


        lst = [i for i in os.listdir(in_folder) if i[-4:] == '.off']
        lst.sort()
        for p in tqdm.tqdm(lst):
            in_path = pjoin(in_folder, p)
            out_path = pjoin(out_folder, p.replace('.off','.pts'))
            if os.path.exists(out_path) and mode == 'train':
                continue
            mesh = trimesh.load(in_path, force='mesh')
            pc, _ = trimesh.sample.sample_surface(mesh, sample_num)
            pc = np.array(pc)
            pc, centroid, scale = pc_normalize(pc)
            np.savetxt(out_path, pc)

            if mode == 'test' and args.fix_test:
                fix_folder = pjoin(args.data_dir, args.category,
                                  mode + '_fix')
                os.makedirs(fix_folder, exist_ok=True)
                fix_path = pjoin(fix_folder, p.replace('.off','.pt')
                                  )
                pc = np.random.permutation(pc)[:1024,:]
                #each instance sample 10 rotations for test
                rgt = tools.
                                  get_sampled_rotation_matrices_by_axisAngle
                                  (10).cpu()
                pc = torch.bmm(rgt, torch.Tensor(pc).unsqueeze(0).
                                  repeat(10,1,1).transpose(2,1))
                data_dict = {'pc':pc.transpose(1,2), 'rgt':rgt,'
                                  centroid':centroid, 'scale':scale}
                torch.save(data_dict, fix_path)
```

B.1.4. test.py

```
import torch
import numpy as np
import random
import os
from os.path import join as pjoin
import sys
import argparse
import pandas as pd

BASEPATH = os.path.dirname(__file__)
sys.path.insert(0,pjoin(BASEPATH, '../..'))
sys.path.insert(0,pjoin(BASEPATH, '..'))
import config as Config
from visualize import visualize
import utils.tools as tools
from model import Model

def test(test_folder, model):
    seed = 1
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)

    geodesic_errors_lst = np.array([])
    l = 0
    test_path_list = [os.path.join(test_folder, i) for i in os.
                                  listdir(test_folder)]
    for i in range(len(test_path_list)):
        path = test_path_list[i]
        tmp = torch.load(path)
```

```python
        pc2 = tmp['pc'].cpu().cuda()
        gt_rmat = tmp['rgt'].cpu().cuda()
        out_rmat, out_nd = model(pc2.transpose(1, 2))
        l += ((gt_rmat - out_rmat) ** 2).sum()
        geodesic_errors = np.array(
            tools.compute_geodesic_distance_from_two_matrices(gt_rmat, out_rmat).data.tolist()) # batch
        geodesic_errors = geodesic_errors / np.pi * 180
        geodesic_errors_lst = np.append(geodesic_errors_lst, geodesic_errors)
    l /= len(test_path_list)

    return geodesic_errors_lst, l


if __name__ == "__main__":
    arg_parser = argparse.ArgumentParser()
    arg_parser.add_argument("--config", type=str, required=True, help="Path to config")
    arg_parser.add_argument("--rotation_map",type=str,required=True, help = 'add rotation representation')
    arg_parser.add_argument("-c", "--checkpoint", type=int, default=-1, help="checkpoint number")
    args = arg_parser.parse_args()

    param=Config.Parameters()
    param.read_config(pjoin("../configs", args.config))

    test_folder = pjoin(param.data_folder, 'test_fix')
    if args.checkpoint == -1:
        allcp = os.listdir(param.write_weight_folder)
        allcp.sort()
        weight_path = pjoin(param.write_weight_folder, allcp[-1])
    else:
        weight_path = pjoin(param.write_weight_folder, "model_%07d.weight"%args.checkpoint)

    with torch.no_grad():
        model = Model(out_rotation_mode=param.out_rotation_mode)
        print("Load " + weight_path)
        f = torch.load(weight_path)
        model.load_state_dict(f['model'])
        model.cuda()
        model.eval()
        errors, l = test(test_folder, model)
    np.save(param.write_weight_folder.replace('/weight',''), errors)
    loss = l
    min_error = np.round(np.min(errors),2)
    Q1= np.round(np.percentile(errors,25),2)
    median_error= np.round(np.percentile(errors,50),2)
    Q3= np.round(np.percentile(errors,75),2)
    mean_error = np.round(errors.mean(), 2)
    max_error = np.round(errors.max(), 2)
    std = np.round(np.std(errors), 2)
    geo_1_deg_error= np.round((errors<1).sum()/len(errors),3)
    geo_3_deg_error= np.round((errors < 3).sum() / len(errors), 3)
    geo_5_deg_error= np.round((errors<5).sum()/len(errors),3)
    representation_map = args.rotation_map
    loss = np.array([l.cpu()][0])
    min_error = np.array([min_error])
    Q1 = np.array([Q1])
    median_error = np.array([median_error])
    Q3 = np.array([Q3])
    max_error = np.array([max_error])
    std = np.array([std])
    geo_1_deg_error = np.array([geo_1_deg_error])
    geo_3_deg_error = np.array([geo_3_deg_error])
    geo_5_deg_error = np.array([geo_5_deg_error])
    data = {'rotation map':representation_map,'loss':loss,
      'min_error':min_error,'Q1':Q1,
      'median_error': median_error, 'Q3':Q3,
      'max': max_error,'std': std,'geo_1_deg_error': geo_1_deg_error
                                                    ,
      'geo_3_deg_error': geo_3_deg_error,'geo_5_deg_error':
                                                    geo_5_deg_error};pd.set_option('
                                                    display.max_colwidth', None)
    Table = pd.DataFrame(data)
    print(Table)
    Table.to_excel("{}.xlsx".format(args.rotation_map),sheet_name =
                                                    args.rotation_map)
```

B.1.5. train.py

```
import torch
import numpy as np
import os
from os.path import join as pjoin
import argparse
import sys

BASEPATH = os.path.dirname(__file__)
sys.path.insert(0,pjoin(BASEPATH, '../..'))
sys.path.insert(0,pjoin(BASEPATH, '..'))
import utils.tools as tools
import utils.rpmg as rpmg
import config as Config
from dataset import ModelNetDataset
from model import Model
from test import test
```

```python

def train_one_iteraton(pc, param, model, optimizer, iteration, tau):
    optimizer.zero_grad()
    batch=pc.shape[0]
    point_num = param.sample_num

    ###get training data######
    pc1 = torch.autograd.Variable(pc.float().cuda()) #num*3
    gt_rmat = tools.get_sampled_rotation_matrices_by_axisAngle(batch)#batch*3*3
    gt_rmats = gt_rmat.contiguous().view(batch,1,3,3).expand(batch, point_num, 3,3 ).contiguous().view(-1,3,3)
    pc2 = torch.bmm(gt_rmats, pc1.view(-1,3,1))#(batch*point_num)*3*1
    pc2 = pc2.view(batch, point_num, 3) ##batch,p_num,3

    ###network forward########
    out_rmat,out_nd = model(pc2.transpose(1,2))   #output [batch(*sample_num),3,3]

    ####compute loss##########
    if not param.use_rpmg:
        loss = ((gt_rmat - out_rmat) ** 2).mean()
    else:
        out_9d = rpmg.RPMG.apply(out_nd, tau, param.rpmg_lambda, gt_rmat, iteration)
        # note here L2 loss should be sum! Or it will affect tau.
        loss = ((gt_rmat - out_9d)**2).sum()

        # # flow loss. need to use tau=50
        # loss = ((pc2 - torch.matmul(pc1, out_9d.transpose(-1,-2)))**2).mean()

        # # geodesic loss. need to use tau=1/10 -> 1/2
        # theta = tools.compute_geodesic_distance_from_two_matrices(gt_rmat, out_9d)
        # loss = (theta **2).sum()
    loss.backward()
    optimizer.step()

    if iteration % 100 == 0:
        param.logger.add_scalar('train_loss', loss.item(), iteration)
        if param.use_rpmg:
            param.logger.add_scalar('k', tau, iteration)
            param.logger.add_scalar('lambda', param.rpmg_lambda, iteration)
        param.logger.add_scalar('nd_norm', out_nd.norm(dim=1).mean().item(), iteration)

    return loss


# pc_lst: [point_num*3]
def train(param):

    torch.cuda.set_device(param.device)

    print ("####Initiate model")
    model = Model(out_rotation_mode=param.out_rotation_mode).cuda()
    optimizer = torch.optim.Adam(model.parameters(), lr=param.lr)
    if param.start_iteration != 0:
        read_path = pjoin(param.write_weight_folder, "model_%07d.
                                  weight"%param.start_iteration)
        print("Load " + read_path)
        checkpoint = torch.load(read_path)
        model.load_state_dict(checkpoint['model'])
        optimizer.load_state_dict(checkpoint['optimizer'])
        start_iteration = checkpoint['iteration']
    else:
        print('start from beginning')
        start_iteration = param.start_iteration

    print ("start train")
    train_folder = os.path.join(param.data_folder, 'train_pc')
    val_folder = os.path.join(param.data_folder, 'test_fix')
    train_dataset = ModelNetDataset(train_folder, sample_num=param.
                                  sample_num)

    train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=param.batch,
        shuffle=True,
        num_workers=4,
        pin_memory=True
    )

    iteration = start_iteration
    while True:
        for data in train_loader:
            model.train()

            #lr decay
            lr = max(param.lr * (0.7 ** (iteration // (param.
                                  total_iteration//10))), 1e-5)
            for param_group in optimizer.param_groups:
                param_group['lr'] = lr

            iteration += 1
            if param.rpmg_tau_strategy == 1:
                tau = 1/4
            elif param.rpmg_tau_strategy == 2:
                tau = 1/20
            elif param.rpmg_tau_strategy == 3:
                tau = 1 / 20 + (1 / 4 - 1 / 20) / 9 * min(iteration
                                  // (param.total_iteration//10), 9)
            elif param.rpmg_tau_strategy == 4:
                tau = -1
            elif param.rpmg_tau_strategy == 5:
                tau = 1 / 10 + (1 / 2 - 1 / 10) / 9 * min(iteration
                                  // (param.total_iteration//10), 9)
            elif param.rpmg_tau_strategy == 6:
                tau = 50
            train_loss = train_one_iteraton(data,  param, model,
                                  optimizer, iteration, tau)
            if (iteration % param.save_weight_iteration == 0):
                print("############# Iteration " + str(iteration) +
                                  " #####################")
                print('train loss: ' + str(train_loss.item()))

                model.eval()
                with torch.no_grad():
                    angle_list, val_loss = test(val_folder, model)
                print('val loss: ' + str( val_loss.item()) )
                param.logger.add_scalar('val_loss', val_loss.item(),
                                   iteration)
                param.logger.add_scalar('val_median',np.median(
                                  angle_list),iteration)
                param.logger.add_scalar('val_mean', angle_list.mean
                                  (),iteration)
                param.logger.add_scalar('val_max', angle_list.max(),
                                  iteration)
                param.logger.add_scalar('val_5accuracy', (angle_list
                                   < 5).sum()/len(angle_list),
                                  iteration)
                param.logger.add_scalar('val_3accuracy', (angle_list
                                   < 3).sum() / len(angle_list),
                                  iteration)
                param.logger.add_scalar('val_1accuracy', (angle_list
                                   < 1).sum() / len(angle_list),
                                  iteration)
                param.logger.add_scalar('lr', lr, iteration)

                path = pjoin(param.write_weight_folder, "model_%07d.
                                  weight"%iteration)
                state = {'model': model.state_dict(), 'optimizer':
                                  optimizer.state_dict(), 'iteration
                                  ': iteration}
                torch.save(state, path)

        if iteration >= param.total_iteration:
            break
```

```

if __name__ == "__main__":

    arg_parser = argparse.ArgumentParser()
    arg_parser.add_argument("--config", type=str, required=True,
                            help="Path to config")
    args = arg_parser.parse_args()

    param=Config.Parameters()
    param.read_config(pjoin("../configs", args.config))

    print(f'use RPMG: {param.use_rpmg}')
    print(f'lambda = {param.rpmg_lambda}')
    if param.rpmg_tau_strategy == 1:
        print('Tau = 1/4')
    elif param.rpmg_tau_strategy == 2:
        print('Tau = 1/20')
    elif param.rpmg_tau_strategy == 3:
        print('Tau = 1/20->1/4')
    elif param.rpmg_tau_strategy == 4:
        print('Tau = gt')
    elif param.rpmg_tau_strategy == 5:
        print('Tau = 1/10->1/2')
    elif param.rpmg_tau_strategy == 6:
        print('Tau = 50')
    rpmg.logger_init(param.logger)
    os.makedirs(param.write_weight_folder, exist_ok=True)

    train(param)
```

B.2. RPMG/ModelNet_PC/configs/

B.2.1. example.config

```
[Record]
exp_folder: ../exps/9D_RPMG_L2
data_folder: ../dataset/modelnet40/airplane

[Params]
lr: 0.001
start_iteration: 0
total_iteration: 30000
save_weight_iteration: 1000

# chocies=["ortho6d", "Quaternion", "svd9d", "axisangle", "euler",
    "10d"]
out_rotation_mode:

```

```
# chocies=[0, 1]. help = "our RPMG only support ortho6d, Quaternion,
    svd9d and 10d!"
use_rpmg:

# # chocies=[1, 2, 3, 4, 5, 6] help= "1,2,3 is for L2 loss. 4 is for
    Tsau_gt. 5 is for geodesic loss. 6 is for flow loss. For
    specific strategies, please see train.py"
rpmg_tau_strategy: 3
rpmg_lambda:
batch:20
sample_num:1024
device: 0
```

B.3. RPMG/ModelNet_PC/pointnet_lib/

B.3.1. pointnet2_modules.py

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import sys
import os
BASEPATH = os.path.dirname(__file__)
sys.path.insert(0, BASEPATH)

CUDA = torch.cuda.is_available()
if CUDA:
    import pointnet2_utils as futils


def knn_point(k, pos2, pos1):
    '''
    Input:
        k: int32, number of k in k-nn search
        pos1: (batch_size, ndataset, c) float32 array, input points
        pos2: (batch_size, npoint, c) float32 array, query points
    Output:
        val: (batch_size, npoint, k) float32 array, L2 distances
        idx: (batch_size, npoint, k) int32 array, indices to input
                                    points
    '''
    if CUDA:
        val, idx = futils.knn(k, pos2, pos1)
        return val, idx.long()

    B, N, C = pos1.shape
    M = pos2.shape[1]
    pos1 = pos1.view(B, 1, N, -1).repeat(1, M, 1, 1)
    pos2 = pos2.view(B, M, 1, -1).repeat(1, 1, N, 1)
    dist = torch.sum(-(pos1 - pos2) ** 2, -1)
    val, idx = dist.topk(k=k, dim=-1)
    return torch.sqrt(-val), idx


def three_nn(xyz1, xyz2):
    if CUDA:
        dists, idx = futils.three_nn(xyz1, xyz2)
        return dists, idx.long()

    dists = square_distance(xyz1, xyz2)
    dists, idx = dists.sort(dim=-1)
    dists, idx = dists[:, :, :3], idx[:, :, :3]  # [B, N, 3]
    return dists, idx


def three_interpolate(points, idx, weight):  # points: [B, C, M],
                                         idx: [B, N, 3], returns [B, C, N]
    if CUDA:
        return futils.three_interpolate(points, idx.int(), weight)

    B, N = idx.shape[:2]
    points = points.permute(0, 2, 1)  # [B, M, C] --> [B, N, 3, C]
    interpolated_points = torch.sum(index_points(points, idx) *
                                    weight.view(B, N, 3, 1), dim=2)
    return interpolated_points.permute(0, 2, 1)


def square_distance(src, dst):
    """
    Calculate Euclid distance between each two points.
    src^T * dst = xn * xm + yn * ym + zn *  z m
    sum(src^2, dim=-1) = xn*xn + yn*yn + zn*zn;
    sum(dst^2, dim=-1) = xm*xm + ym*ym + zm*zm;
    dist = (xn - xm)^2 + (yn - ym)^2 + (zn - zm)^2
         = sum(src**2,dim=-1)+sum(dst**2,dim=-1)-2*src^T*dst
    Input:
        src: source points, [B, N, C]
        dst: target points, [B, M, C]
    Output:
        dist: per-point square distance, [B, N, M]
    """
    B, N, _ = src.shape
    _, M, _ = dst.shape
    dist = -2 * torch.matmul(src, dst.permute(0, 2, 1))
    dist += torch.sum(src ** 2, -1).view(B, N, 1)
    dist += torch.sum(dst ** 2, -1).view(B, 1, M)
    return dist


def index_points(points, idx):
    """
    Input:
        points: input points data, [B, N, C]
        idx: sample index data, [B, S] or [B, S1, S2, ..Sk]
    Return:
        new_points:, indexed points data, [B, S, C] or [B, S1, S2, ..Sk, C]
    """
    device = points.device
    B = points.shape[0]
    view_shape = list(idx.shape)
    view_shape[1:] = [1] * (len(view_shape) - 1)
    repeat_shape = list(idx.shape)
    repeat_shape[0] = 1
    batch_indices = torch.arange(B, dtype=torch.long).to(device).view(view_shape).repeat(repeat_shape)
    new_points = points[batch_indices, idx, :]
    return new_points


def gather_operation(feature, idx):  # [B, C, N], [B, npoint] -> [B, C, npoint]
    if CUDA:
        return futils.gather_operation(feature, idx)
    return index_points(feature.transpose(-1, -2), idx).transpose(-1, -2)


def group_operation(feature, idx):  # [B, C, N], idx [B, npoint, nsample] --> [B, C, npoint, nsample]
    if CUDA:
        return futils.grouping_operation(feature, idx)
    return index_points(feature.transpose(-1, -2), idx).permute(0, 3, 1, 2)


def farthest_point_sample(xyz, npoint):
    """
    Input:
        xyz: pointcloud data, [B, N, 3]
        npoint: number of samples
    Return:
        centroids: sampled pointcloud index, [B, npoint]
    """
    if CUDA:
        idx = futils.furthest_point_sample(xyz, npoint).long()
        return idx

    device = xyz.device
    B, N, C = xyz.shape

    centroids = torch.zeros(B, npoint, dtype=torch.long).to(device)
    distance = torch.ones(B, N).to(device) * 1e10
    farthest = torch.randint(0, N, (B,), dtype=torch.long).to(device
                                  )
    batch_indices = torch.arange(B, dtype=torch.long).to(device)
    for i in range(npoint):
        centroids[:, i] = farthest
        centroid = xyz[batch_indices, farthest, :].view(B, 1, 3)
        dist = torch.sum((xyz - centroid) ** 2, -1)
        mask = dist < distance
        distance[mask] = dist[mask]
        farthest = torch.max(distance, -1)[1]
    return centroids


def query_ball_point(radius, nsample, xyz, new_xyz):
    """
    Input:
        radius: local region radius
        nsample: max sample number in local region
        xyz: all points, [B, N, 3]
        new_xyz: query points, [B, S, 3]
    Return:
        group_idx: grouped points index, [B, S, nsample]
    """
    if CUDA:
        return futils.ball_query(radius, nsample, xyz, new_xyz).long
                                  ()

    device = xyz.device
    B, N, C = xyz.shape
    _, S, _ = new_xyz.shape

    group_idx = torch.arange(N, dtype=torch.long).to(device).view(1,
                                   1, N).repeat([B, S, 1])
    sqrdists = square_distance(new_xyz, xyz)
    group_idx[sqrdists > radius ** 2] = N
    group_idx = group_idx.sort(dim=-1)[0][:, :, :nsample]
    group_first = group_idx[:, :, 0].view(B, S, 1).repeat([1, 1,
                                  nsample])
    mask_first = group_first == N
    group_first[mask_first] = 0
    mask = group_idx == N
    group_idx[mask] = group_first[mask]

    return group_idx


def sample_and_group_all(xyz, points):
```

```python
    """
    Input:
        xyz: input points position data, [B, N, 3]
        points: input points data, [B, N, D]
    Return:
        new_xyz: sampled points position data, [B, 1, 3]
        new_points: sampled points data, [B, 1, N, 3+D]
    """
    device = xyz.device
    B, N, C = xyz.shape
    new_xyz = torch.zeros(B, 1, C).to(device)
    grouped_xyz = xyz.view(B, 1, N, C)
    if points is not None:
        new_points = torch.cat([grouped_xyz, points.view(B, 1, N, -1)], dim=-1)
    else:
        new_points = grouped_xyz
    return new_xyz, new_points


class PointNetSetAbstractionMsg(nn.Module):
    def __init__(self, npoint, radius_list, nsample_list, in_channel, mlp_list, knn=False):
        super(PointNetSetAbstractionMsg, self).__init__()
        self.npoint = npoint
        self.radius_list = radius_list
        self.nsample_list = nsample_list
        self.conv_blocks = nn.ModuleList()
        self.bn_blocks = nn.ModuleList()
        self.out_channel = 0
        for i in range(len(mlp_list)):
            convs = nn.ModuleList()
            bns = nn.ModuleList()
            last_channel = in_channel
            for out_channel in mlp_list[i]:
                convs.append(nn.Conv2d(last_channel, out_channel, 1))
                bns.append(nn.BatchNorm2d(out_channel))
                last_channel = out_channel
            self.out_channel += last_channel
            self.conv_blocks.append(convs)
            self.bn_blocks.append(bns)
        self.knn = knn

    def forward(self, xyz, points):
        """
        Input:
            xyz: input points position data, [B, C, N]
            points: input points data, [B, D, N]
        Return:
            new_xyz: sampled points position data, [B, C, S]
            new_points_concat: sample points feature data, [B, D', S]
        """

        B, C, N = xyz.shape
        S = self.npoint
        fps_idx = farthest_point_sample(xyz.permute(0, 2, 1), S).int()
        new_xyz = gather_operation(xyz, fps_idx)  # [B, C, S]
        new_points_list = []
        for i, radius in enumerate(self.radius_list):
            K = self.nsample_list[i]
            if self.knn:
                _, group_idx = knn_point(K, new_xyz.transpose(-1, -2), xyz.transpose(-1, -2))
            else:
                group_idx = query_ball_point(radius, K, xyz.transpose(-1, -2), new_xyz.transpose(-1, -2))  # [B, S, nsample]
            grouped_xyz = group_operation(xyz, group_idx)  # [B, C, S, nsample]
            grouped_xyz -= new_xyz.view(B, C, S, 1)
            if points is not None:
                grouped_points = group_operation(points, group_idx)  # [B, D, S, nsample]
                grouped_points = torch.cat([grouped_points, grouped_xyz], dim=1)
            else:
                grouped_points = grouped_xyz

            for j in range(len(self.conv_blocks[i])):
                conv = self.conv_blocks[i][j]
                bn = self.bn_blocks[i][j]
                grouped_points = F.relu(bn(conv(grouped_points)))  # [B, D, S, nsample]
            new_points = torch.max(grouped_points, -1)[0]  # [B, D', S]
            new_points_list.append(new_points)

        new_points_concat = torch.cat(new_points_list, dim=1)
        return new_xyz, new_points_concat


class PointNetSetAbstraction(nn.Module):
    def __init__(self, npoint, radius, nsample, in_channel, mlp, group_all, knn=False):
        super(PointNetSetAbstraction, self).__init__()
        self.npoint = npoint
        self.radius = radius
        self.nsample = nsample
```

```python
        self.mlp_convs = nn.ModuleList()
        self.mlp_bns = nn.ModuleList()
        last_channel = in_channel
        for out_channel in mlp:
            self.mlp_convs.append(nn.Conv2d(last_channel, out_channel, 1))
            self.mlp_bns.append(nn.BatchNorm2d(out_channel))
            last_channel = out_channel
        self.out_channel = last_channel
        self.group_all = group_all
        self.knn = knn

    def forward(self, xyz, points):
        """
        Input:
            xyz: input points position data, [B, C, N]
            points: input points data, [B, D, N]
        Return:
            new_xyz: sampled points position data, [B, C, S]
            new_points_concat: sample points feature data, [B, D', S]
        """
        xyz = xyz.permute(0, 2, 1)
        if points is not None:
            points = points.permute(0, 2, 1)
        if self.group_all:
            new_xyz, new_points = sample_and_group_all(xyz, points)
        else:
            assert 0, 'Not Implemented'

        new_points = new_points.permute(0, 3, 2, 1)  # [B, 1, N, 3 + D] --> [B, 3 + D, N, 1]
        for i, conv in enumerate(self.mlp_convs):
            bn = self.mlp_bns[i]
            new_points = F.relu(bn(conv(new_points)))

        new_points = torch.max(new_points, 2)[0]
        new_xyz = new_xyz.permute(0, 2, 1)
        return new_xyz, new_points


class PointNetFeaturePropagation(nn.Module):
    def __init__(self, in_channel, mlp):
        super(PointNetFeaturePropagation, self).__init__()
        self.mlp_convs = nn.ModuleList()
        self.mlp_bns = nn.ModuleList()
        last_channel = in_channel
        for out_channel in mlp:
            self.mlp_convs.append(nn.Conv1d(last_channel, out_channel, 1))
            self.mlp_bns.append(nn.BatchNorm1d(out_channel))
```

```python
                last_channel = out_channel
            self.out_channel = last_channel

        def forward(self, xyz1, xyz2, points1, points2):
            """
            Input:
                xyz1: input points position data, [B, C, N]
                xyz2: sampled input points position data, [B, C, S]
                points1: input points data, [B, D, N]
                points2: input points data, [B, D, S]
            Return:
                new_points: upsampled points data, [B, D', N]
            """
            xyz1 = xyz1.permute(0, 2, 1)
            xyz2 = xyz2.permute(0, 2, 1)

            B, N, C = xyz1.shape
            _, S, _ = xyz2.shape

            if S == 1:
                interpolated_points = points2.repeat(1, 1, N)
            else:
                dist, idx = three_nn(xyz1, xyz2)
                dist_recip = 1.0 / (dist + 1e-8)
                norm = torch.sum(dist_recip, dim=2, keepdim=True)
                weight = dist_recip / norm

                interpolated_points = three_interpolate(points2, idx,
                                      weight)  # [B, C, N]

            if points1 is not None:
                new_points = torch.cat([points1, interpolated_points],
                                      dim=-2)
            else:
                new_points = interpolated_points

            for i, conv in enumerate(self.mlp_convs):
                bn = self.mlp_bns[i]
                new_points = F.relu(bn(conv(new_points)))
            return new_points
```

B.3.2. pointnet2_utils.py

```python
import torch
from torch.autograd import Variable
from torch.autograd import Function
import torch.nn as nn
from typing import Tuple

import pointnet2_cuda as pointnet2
```

```python


class FurthestPointSampling(Function):
    @staticmethod
    def forward(ctx, xyz: torch.Tensor, npoint: int) -> torch.Tensor
                                          :
        """
        Uses iterative furthest point sampling to select a set of
                                          npoint features that have the
                                          largest
        minimum distance
        :param ctx:
        :param xyz: (B, N, 3) where N > npoint
        :param npoint: int, number of features in the sampled set
        :return:
             output: (B, npoint) tensor containing the set
        """
        xyz = xyz.contiguous()
        # assert xyz.is_contiguous()

        B, N, _ = xyz.size()
        output = torch.cuda.IntTensor(B, npoint)
        temp = torch.cuda.FloatTensor(B, N).fill_(1e10)

        pointnet2.furthest_point_sampling_wrapper(B, N, npoint, xyz,
                                           temp, output)
        return output

    @staticmethod
    def backward(xyz, a=None):
        return None, None


furthest_point_sample = FurthestPointSampling.apply


class GatherOperation(Function):

    @staticmethod
    def forward(ctx, features: torch.Tensor, idx: torch.Tensor) ->
                                          torch.Tensor:
        """
        :param ctx:
        :param features: (B, C, N)
        :param idx: (B, npoint) index tensor of the features to
                                          gather
        :return:
            output: (B, C, npoint)
        """
        features = features.contiguous()
        idx = idx.contiguous()
        assert features.is_contiguous()
        assert idx.is_contiguous()

        B, npoint = idx.size()
        _, C, N = features.size()
        output = torch.cuda.FloatTensor(B, C, npoint)

        pointnet2.gather_points_wrapper(B, C, N, npoint, features, idx, output)

        ctx.for_backwards = (idx, C, N)
        return output

    @staticmethod
    def backward(ctx, grad_out):
        idx, C, N = ctx.for_backwards
        B, npoint = idx.size()

        grad_features = Variable(torch.cuda.FloatTensor(B, C, N).zero_())
        grad_out_data = grad_out.data.contiguous()
        pointnet2.gather_points_grad_wrapper(B, C, N, npoint, grad_out_data, idx, grad_features.data)
        return grad_features, None


gather_operation = GatherOperation.apply

class KNN(Function):

    @staticmethod
    def forward(ctx, k: int, unknown: torch.Tensor, known: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        """
        Find the three nearest neighbors of unknown in known
        :param ctx:
        :param unknown: (B, N, 3)
        :param known: (B, M, 3)
        :return:
            dist: (B, N, k) l2 distance to the three nearest neighbors
            idx: (B, N, k) index of 3 nearest neighbors
        """
        unknown = unknown.contiguous()
        known = known.contiguous()
        assert unknown.is_contiguous()
        assert known.is_contiguous()

        B, N, _ = unknown.size()
        m = known.size(1)
        dist2 = torch.cuda.FloatTensor(B, N, k)
        idx = torch.cuda.IntTensor(B, N, k)

        pointnet2.knn_wrapper(B, N, m, k, unknown, known, dist2, idx
                                 )
        return torch.sqrt(dist2), idx

    @staticmethod
    def backward(ctx, a=None, b=None):
        return None, None, None

knn = KNN.apply

class ThreeNN(Function):

    @staticmethod
    def forward(ctx, unknown: torch.Tensor, known: torch.Tensor) ->
                                 Tuple[torch.Tensor, torch.Tensor]:
        """
        Find the three nearest neighbors of unknown in known
        :param ctx:
        :param unknown: (B, N, 3)
        :param known: (B, M, 3)
        :return:
            dist: (B, N, 3) l2 distance to the three nearest
                                 neighbors
            idx: (B, N, 3) index of 3 nearest neighbors
        """
        unknown = unknown.contiguous()
        known = known.contiguous()
        assert unknown.is_contiguous()
        assert known.is_contiguous()

        B, N, _ = unknown.size()
        m = known.size(1)
        dist2 = torch.cuda.FloatTensor(B, N, 3)
        idx = torch.cuda.IntTensor(B, N, 3)

        pointnet2.three_nn_wrapper(B, N, m, unknown, known, dist2,
                                 idx)
        return torch.sqrt(dist2), idx

    @staticmethod
    def backward(ctx, a=None, b=None):
        return None, None


three_nn = ThreeNN.apply


```

```python
class ThreeInterpolate(Function):

    @staticmethod
    def forward(ctx, features: torch.Tensor, idx: torch.Tensor,
                                weight: torch.Tensor) -> torch.
                                Tensor:
        """
        Performs weight linear interpolation on 3 features
        :param ctx:
        :param features: (B, C, M) Features descriptors to be
                                interpolated from
        :param idx: (B, n, 3) three nearest neighbors of the target
                                features in features
        :param weight: (B, n, 3) weights
        :return:
            output: (B, C, N) tensor of the interpolated features
        """
        features = features.contiguous()
        idx = idx.contiguous()
        weight = weight.contiguous()
        assert features.is_contiguous()
        assert idx.is_contiguous()
        assert weight.is_contiguous()

        B, c, m = features.size()
        n = idx.size(1)
        ctx.three_interpolate_for_backward = (idx, weight, m)
        output = torch.cuda.FloatTensor(B, c, n)

        pointnet2.three_interpolate_wrapper(B, c, m, n, features,
                                idx, weight, output)
        return output

    @staticmethod
    def backward(ctx, grad_out: torch.Tensor) -> Tuple[torch.Tensor,
                                torch.Tensor, torch.Tensor]:
        """
        :param ctx:
        :param grad_out: (B, C, N) tensor with gradients of outputs
        :return:
            grad_features: (B, C, M) tensor with gradients of
                                features
            None:
            None:
        """
        idx, weight, m = ctx.three_interpolate_for_backward
        B, c, n = grad_out.size()

        grad_features = Variable(torch.cuda.FloatTensor(B, c, m).
                                zero_())
        grad_out_data = grad_out.data.contiguous()

        pointnet2.three_interpolate_grad_wrapper(B, c, n, m,
                                grad_out_data, idx, weight,
                                grad_features.data)
        return grad_features, None, None


three_interpolate = ThreeInterpolate.apply


class GroupingOperation(Function):

    @staticmethod
    def forward(ctx, features: torch.Tensor, idx: torch.Tensor) ->
                                torch.Tensor:
        """
        :param ctx:
        :param features: (B, C, N) tensor of features to group
        :param idx: (B, npoint, nsample) tensor containing the
                                indicies of features to group with
        :return:
            output: (B, C, npoint, nsample) tensor
        """
        features = features.contiguous()
        idx = idx.contiguous()
        assert features.is_contiguous()
        assert idx.is_contiguous()
        idx = idx.int()
        B, nfeatures, nsample = idx.size()
        _, C, N = features.size()
        output = torch.cuda.FloatTensor(B, C, nfeatures, nsample)

        pointnet2.group_points_wrapper(B, C, N, nfeatures, nsample,
                                features, idx, output)

        ctx.for_backwards = (idx, N)
        return output

    @staticmethod
    def backward(ctx, grad_out: torch.Tensor) -> Tuple[torch.Tensor,
                                torch.Tensor]:
        """
        :param ctx:
        :param grad_out: (B, C, npoint, nsample) tensor of the
                                gradients of the output from
                                forward
        :return:
            grad_features: (B, C, N) gradient of the features
        """
        idx, N = ctx.for_backwards

        B, C, npoint, nsample = grad_out.size()
        grad_features = Variable(torch.cuda.FloatTensor(B, C, N).
                                  zero_())

        grad_out_data = grad_out.data.contiguous()
        pointnet2.group_points_grad_wrapper(B, C, N, npoint, nsample
                                  , grad_out_data, idx,
                                  grad_features.data)
        return grad_features, None


grouping_operation = GroupingOperation.apply


class BallQuery(Function):

    @staticmethod
    def forward(ctx, radius: float, nsample: int, xyz: torch.Tensor,
                                   new_xyz: torch.Tensor) -> torch.
                                  Tensor:
        """
        :param ctx:
        :param radius: float, radius of the balls
        :param nsample: int, maximum number of features in the balls
        :param xyz: (B, N, 3) xyz coordinates of the features
        :param new_xyz: (B, npoint, 3) centers of the ball query
        :return:
            idx: (B, npoint, nsample) tensor with the indicies of
                                  the features that form the query
                                  balls
        """
        new_xyz = new_xyz.contiguous()
        xyz = xyz.contiguous()
        assert new_xyz.is_contiguous()
        assert xyz.is_contiguous()

        B, N, _ = xyz.size()
        npoint = new_xyz.size(1)
        idx = torch.cuda.IntTensor(B, npoint, nsample).zero_()

        pointnet2.ball_query_wrapper(B, N, npoint, radius, nsample,
                                  new_xyz, xyz, idx)
        return idx

    @staticmethod
    def backward(ctx, a=None):
        return None, None, None, None


ball_query = BallQuery.apply

```

```

class QueryAndGroup(nn.Module):
    def __init__(self, radius: float, nsample: int, use_xyz: bool = True):
        """
        :param radius: float, radius of ball
        :param nsample: int, maximum number of features to gather in the ball
        :param use_xyz:
        """
        super().__init__()
        self.radius, self.nsample, self.use_xyz = radius, nsample, use_xyz

    def forward(self, xyz: torch.Tensor, new_xyz: torch.Tensor, features: torch.Tensor = None) -> Tuple[torch.Tensor]:
        """
        :param xyz: (B, N, 3) xyz coordinates of the features
        :param new_xyz: (B, npoint, 3) centroids
        :param features: (B, C, N) descriptors of the features
        :return:
            new_features: (B, 3 + C, npoint, nsample)
        """
        idx = ball_query(self.radius, self.nsample, xyz, new_xyz)
        xyz_trans = xyz.transpose(1, 2).contiguous()
        grouped_xyz = grouping_operation(xyz_trans, idx)  # (B, 3, npoint, nsample)
        grouped_xyz -= new_xyz.transpose(1, 2).unsqueeze(-1)

        if features is not None:
            grouped_features = grouping_operation(features, idx)
            if self.use_xyz:
                new_features = torch.cat([grouped_features, grouped_xyz] , dim=1)  # (B, C + 3, npoint, nsample)
            else:
                new_features = grouped_features
        else:
            assert self.use_xyz, "Cannot have not features and not use xyz as a feature!"
            new_features = grouped_xyz

        return new_features


class GroupAll(nn.Module):
    def __init__(self, use_xyz: bool = True):
        super().__init__()
        self.use_xyz = use_xyz

    def forward(self, xyz: torch.Tensor, new_xyz: torch.Tensor,
                                        features: torch.Tensor = None):
        """
        :param xyz: (B, N, 3) xyz coordinates of the features
        :param new_xyz: ignored
        :param features: (B, C, N) descriptors of the features
        :return:
            new_features: (B, C + 3, 1, N)
        """
        grouped_xyz = xyz.transpose(1, 2).unsqueeze(2)
        if features is not None:
            grouped_features = features.unsqueeze(2)
            if self.use_xyz:
                new_features = torch.cat([grouped_xyz,
                                        grouped_features], dim=1)  # (B, 3
                                         + C, 1, N)
            else:
                new_features = grouped_features
        else:
            new_features = grouped_xyz

        return new_features


class KNNAndGroup(nn.Module):
    def __init__(self, radius:float, nsample: int, use_xyz: bool =
                                        True):
        """
        :param radius: float, radius of ball
        :param nsample: int, maximum number of features to gather in
                                         the ball
        :param use_xyz:
        """
        super().__init__()
        self.radius, self.nsample, self.use_xyz = radius, nsample,
                                        use_xyz

    def forward(self, xyz: torch.Tensor, new_xyz: torch.Tensor =
                                        None, idx: torch.Tensor = None,
                                        features: torch.Tensor = None) ->
                                        Tuple[torch.Tensor]:
        """
        :param xyz: (B, N, 3) xyz coordinates of the features
        :param new_xyz: (B, M, 3) centroids
        :param idx: (B, M, K) centroids
        :param features: (B, C, N) descriptors of the features
        :return:
            new_features: (B, 3 + C, M, K) if use_xyz = True else (B
                                        , C, M, K)
        """

        ##TODO: implement new_xyz into knn
        if new_xyz is None:
            new_xyz = xyz

        if idx is None:
            idx = knn(xyz, new_xyz, self.radius, self.nsample) # B, M, K
        idx = idx.detach()

        xyz_trans = xyz.transpose(1, 2).contiguous()
        new_xyz_trans = new_xyz.transpose(1, 2).contiguous()

        grouped_xyz = grouping_operation(xyz_trans, idx) # B, 3, M, K
        grouped_xyz -= new_xyz_trans.unsqueeze(-1) # B, 3, M, K
        #grouped_r = torch.norm(grouped_xyz, dim=1).max(dim=-1)[0]#B,M
        #print(new_xyz.shape[1], grouped_r)

        if features is not None:
            grouped_features = grouping_operation(features, idx) # B, C, M, K
            # grouped_features_test = grouping_operation(features, idx)
            # assert (grouped_features == grouped_features).all()
            if self.use_xyz:
                new_features = torch.cat([grouped_xyz, grouped_features], dim=1)  # (B, C + 3, M, K)
            else:
                new_features = grouped_features
        else:
            assert self.use_xyz, "Cannot have not features and not use xyz as a feature!"
            new_features = grouped_xyz

        return new_features
```

B.4. RPMG/ModelNet_PC/

B.4.1. model.py

```
import torch
import torch.nn as nn
import sys
import os
from os.path import join as pjoin

BASEPATH = os.path.dirname(__file__)
```

```python
sys.path.insert(0,pjoin(BASEPATH, '../..'))
import utils.tools as tools
from pointnets import PointNet2_cls

class Model(nn.Module):
    def __init__(self, out_rotation_mode="Quaternion"):
        super(Model, self).__init__()

        self.out_rotation_mode = out_rotation_mode

        if(out_rotation_mode == "Quaternion"):
            self.out_channel = 4
        elif (out_rotation_mode  == "ortho6d"):
            self.out_channel = 6
        elif (out_rotation_mode  == "svd9d"):
            self.out_channel = 9
        elif (out_rotation_mode  == "10d"):
            self.out_channel = 10
        elif out_rotation_mode == 'euler':
            self.out_channel = 3
        elif out_rotation_mode == 'axisangle':
            self.out_channel = 4
        else:
            raise NotImplementedError

        print(out_rotation_mode)

        self.model = PointNet2_cls(self.out_channel)


    #pt b*point_num*3
    def forward(self, input):
        out_nd = self.model(input)

        if(self.out_rotation_mode == "Quaternion"):
            out_rmat = tools.compute_rotation_matrix_from_quaternion
                                  (out_nd) #b*3*3
        elif(self.out_rotation_mode=="ortho6d"):
            out_rmat = tools.compute_rotation_matrix_from_ortho6d(
                                  out_nd) #b*3*3
        elif(self.out_rotation_mode=="svd9d"):
            out_rmat = tools.symmetric_orthogonalization(out_nd)  #
                                  b*3*3
        elif (self.out_rotation_mode == "10d"):
            out_rmat = tools.compute_rotation_matrix_from_10d(out_nd
                                  )  # b*3*3
        elif (self.out_rotation_mode == "euler"):
            out_rmat = tools.compute_rotation_matrix_from_euler(
                                  out_nd)  # b*3*3
        elif (self.out_rotation_mode == "axisangle"):
            out_rmat = tools.compute_rotation_matrix_from_axisAngle(
                                                out_nd)  # b*3*3

        return out_rmat, out_nd
```

B.4.2. pointnet_utils.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from time import time
import numpy as np


def timeit(tag, t):
    print("{}: {}s".format(tag, time() - t))
    return time()

def square_distance(src, dst):
    """
    Calculate Euclid distance between each two points.
    src^T * dst = xn * xm + yn * ym + zn *  z m
    sum(src^2, dim=-1) = xn*xn + yn*yn + zn*zn;
    sum(dst^2, dim=-1) = xm*xm + ym*ym + zm*zm;
    dist = (xn - xm)^2 + (yn - ym)^2 + (zn - zm)^2
         = sum(src**2,dim=-1)+sum(dst**2,dim=-1)-2*src^T*dst
    Input:
        src: source points, [B, N, C]
        dst: target points, [B, M, C]
    Output:
        dist: per-point square distance, [B, N, M]
    """
    B, N, _ = src.shape
    _, M, _ = dst.shape
    dist = -2 * torch.matmul(src, dst.permute(0, 2, 1))
    dist += torch.sum(src ** 2, -1).view(B, N, 1)
    dist += torch.sum(dst ** 2, -1).view(B, 1, M)
    return dist


def index_points(points, idx):
    """
    Input:
        points: input points data, [B, N, C]
        idx: sample index data, [B, S]
    Return:
        new_points:, indexed points data, [B, S, C]
    """
    device = points.device
    B = points.shape[0]
    view_shape = list(idx.shape)
    view_shape[1:] = [1] * (len(view_shape) - 1)
    repeat_shape = list(idx.shape)
    repeat_shape[0] = 1
    batch_indices = torch.arange(B, dtype=torch.long).to(device).
                              view(view_shape).repeat(
                              repeat_shape)
    new_points = points[batch_indices, idx, :]
    return new_points


def farthest_point_sample(xyz, npoint):
    """
    Input:
        xyz: pointcloud data, [B, N, 3]
        npoint: number of samples
    Return:
        centroids: sampled pointcloud index, [B, npoint]
    """
    device = xyz.device
    B, N, C = xyz.shape
    centroids = torch.zeros(B, npoint, dtype=torch.long).to(device)
    distance = torch.ones(B, N).to(device) * 1e10
    farthest = torch.randint(0, N, (B,), dtype=torch.long).to(device
                              )
    batch_indices = torch.arange(B, dtype=torch.long).to(device)
    for i in range(npoint):
        centroids[:, i] = farthest
        centroid = xyz[batch_indices, farthest, :].view(B, 1, 3)
        dist = torch.sum((xyz - centroid) ** 2, -1)
        mask = dist < distance
        distance[mask] = dist[mask]
        farthest = torch.max(distance, -1)[1]
    return centroids


def query_ball_point(radius, nsample, xyz, new_xyz):
    """
    Input:
        radius: local region radius
        nsample: max sample number in local region
        xyz: all points, [B, N, 3]
        new_xyz: query points, [B, S, 3]
    Return:
        group_idx: grouped points index, [B, S, nsample]
    """
    device = xyz.device
    B, N, C = xyz.shape
    _, S, _ = new_xyz.shape
    group_idx = torch.arange(N, dtype=torch.long).to(device).view(1,
                              1, N).repeat([B, S, 1])
```

```python
    sqrdists = square_distance(new_xyz, xyz)
    group_idx[sqrdists > radius ** 2] = N
    group_idx = group_idx.sort(dim=-1)[0][:, :, :nsample]
    group_first = group_idx[:, :, 0].view(B, S, 1).repeat([1, 1, nsample])
    mask = group_idx == N
    group_idx[mask] = group_first[mask]
    return group_idx


def sample_and_group(npoint, radius, nsample, xyz, points, returnfps=False):
    """
    Input:
        npoint:
        radius:
        nsample:
        xyz: input points position data, [B, N, 3]
        points: input points data, [B, N, D]
    Return:
        new_xyz: sampled points position data, [B, npoint, nsample, 3]
        new_points: sampled points data, [B, npoint, nsample, 3+D]
    """
    B, N, C = xyz.shape
    S = npoint
    fps_idx = farthest_point_sample(xyz, npoint)  # [B, npoint, C]
    new_xyz = index_points(xyz, fps_idx)
    idx = query_ball_point(radius, nsample, xyz, new_xyz)
    grouped_xyz = index_points(xyz, idx)  # [B, npoint, nsample, C]
    grouped_xyz_norm = grouped_xyz - new_xyz.view(B, S, 1, C)

    if points is not None:
        grouped_points = index_points(points, idx)
        new_points = torch.cat([grouped_xyz_norm, grouped_points], dim=-1)  # [B, npoint, nsample, C+D]
    else:
        new_points = grouped_xyz_norm
    if returnfps:
        return new_xyz, new_points, grouped_xyz, fps_idx
    else:
        return new_xyz, new_points


def sample_and_group_all(xyz, points):
    """
    Input:
        xyz: input points position data, [B, N, 3]
        points: input points data, [B, N, D]
    Return:
        new_xyz: sampled points position data, [B, 1, 3]
        new_points: sampled points data, [B, 1, N, 3+D]
    """
    device = xyz.device
    B, N, C = xyz.shape
    new_xyz = torch.zeros(B, 1, C).to(device)
    grouped_xyz = xyz.view(B, 1, N, C)
    if points is not None:
        new_points = torch.cat([grouped_xyz, points.view(B, 1, N, -1
                                )], dim=-1)
    else:
        new_points = grouped_xyz
    return new_xyz, new_points


class PointNetSetAbstraction(nn.Module):
    def __init__(self, npoint, radius, nsample, in_channel, mlp,
                                group_all):
        super(PointNetSetAbstraction, self).__init__()
        self.npoint = npoint
        self.radius = radius
        self.nsample = nsample
        self.mlp_convs = nn.ModuleList()
        self.mlp_bns = nn.ModuleList()
        last_channel = in_channel
        for out_channel in mlp:
            self.mlp_convs.append(nn.Conv2d(last_channel,
                                out_channel, 1))
            self.mlp_bns.append(nn.BatchNorm2d(out_channel))
            last_channel = out_channel
        self.group_all = group_all

    def forward(self, xyz, points):
        """
        Input:
            xyz: input points position data, [B, C, N]
            points: input points data, [B, D, N]
        Return:
            new_xyz: sampled points position data, [B, C, S]
            new_points_concat: sample points feature data, [B, D', S
                                ]
        """
        xyz = xyz.permute(0, 2, 1)
        if points is not None:
            points = points.permute(0, 2, 1)

        if self.group_all:
            new_xyz, new_points = sample_and_group_all(xyz, points)
        else:
            new_xyz, new_points = sample_and_group(self.npoint, self
                                .radius, self.nsample, xyz, points
```

```
                                  )
        # new_xyz: sampled points position data, [B, npoint, C]
        # new_points: sampled points data, [B, npoint, nsample, C+D]
        new_points = new_points.permute(0, 3, 2, 1)  # [B, C+D,
                                  nsample,npoint]
        for i, conv in enumerate(self.mlp_convs):
            bn = self.mlp_bns[i]
            new_points = F.relu(bn(conv(new_points)), inplace=True)

        new_points = torch.max(new_points, 2)[0]
        new_xyz = new_xyz.permute(0, 2, 1)
        return new_xyz, new_points


class PointNetSetAbstractionMsg(nn.Module):
    def __init__(self, npoint, radius_list, nsample_list, in_channel
                                  , mlp_list):
        super(PointNetSetAbstractionMsg, self).__init__()
        self.npoint = npoint
        self.radius_list = radius_list
        self.nsample_list = nsample_list
        self.conv_blocks = nn.ModuleList()
        self.bn_blocks = nn.ModuleList()
        for i in range(len(mlp_list)):
            convs = nn.ModuleList()
            bns = nn.ModuleList()
            last_channel = in_channel + 3
            for out_channel in mlp_list[i]:
                convs.append(nn.Conv2d(last_channel, out_channel, 1)
                                  )
                bns.append(nn.BatchNorm2d(out_channel))
                last_channel = out_channel
            self.conv_blocks.append(convs)
            self.bn_blocks.append(bns)

    def forward(self, xyz, points):
        """
        Input:
            xyz: input points position data, [B, C, N]
            points: input points data, [B, D, N]
        Return:
            new_xyz: sampled points position data, [B, C, S]
            new_points_concat: sample points feature data, [B, D', S
                                  ]
        """
        xyz = xyz.permute(0, 2, 1)
        if points is not None:
            points = points.permute(0, 2, 1)

        B, N, C = xyz.shape
        S = self.npoint
        new_xyz = index_points(xyz, farthest_point_sample(xyz, S))
        new_points_list = []
        for i, radius in enumerate(self.radius_list):
            K = self.nsample_list[i]
            group_idx = query_ball_point(radius, K, xyz, new_xyz)
            grouped_xyz = index_points(xyz, group_idx)
            grouped_xyz -= new_xyz.view(B, S, 1, C)
            if points is not None:
                grouped_points = index_points(points, group_idx)
                grouped_points = torch.cat([grouped_points,
                                  grouped_xyz], dim=-1)
            else:
                grouped_points = grouped_xyz

            grouped_points = grouped_points.permute(0, 3, 2, 1)  # [
                                  B, D, K, S]
            for j in range(len(self.conv_blocks[i])):
                conv = self.conv_blocks[i][j]
                bn = self.bn_blocks[i][j]
                grouped_points = F.relu(bn(conv(grouped_points)),
                                  inplace=True)
            new_points = torch.max(grouped_points, 2)[0]  # [B, D',
                                  S]
            new_points_list.append(new_points)

        new_xyz = new_xyz.permute(0, 2, 1)
        new_points_concat = torch.cat(new_points_list, dim=1)
        return new_xyz, new_points_concat
```

B.4.3. pointnets.py

```
import torch.nn as nn
import torch
import torch.nn.functional as F
import os
import sys
BASEPATH = os.path.dirname(__file__)
sys.path.insert(0, BASEPATH)
from pointnet_utils import PointNetSetAbstractionMsg,
                                  PointNetSetAbstraction


class PointNet(nn.Module):
    def __init__(self, out_channel):
        super(PointNet, self).__init__()
        self.feature_extracter = nn.Sequential(
            nn.Conv1d(3, 64, kernel_size=1),
            nn.LeakyReLU(),
            nn.Conv1d(64, 128, kernel_size=1),
            nn.LeakyReLU(),
```

```python
            nn.Conv1d(128, 1024, kernel_size=1),
            nn.AdaptiveMaxPool1d(output_size=1)
        )

        self.mlp = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(),
            nn.Linear(512, out_channel))

    def forward(self, x):
        batch = x.shape[0]
        x = self.feature_extracter(x).view(batch, -1)
        out_data = self.mlp(x)
        return out_data


class PointNet2_MSG(nn.Module):
    def __init__(self, out_channel):
        super(PointNet2_MSG, self).__init__()
        self.sa1 = PointNetSetAbstractionMsg(512, [0.1, 0.2, 0.4], [
                                             32, 64, 128], 3, [[32, 32, 64], [
                                             64, 64, 128], [64, 96, 128]])
        self.sa2 = PointNetSetAbstractionMsg(128, [0.4,0.8], [64,
                                             128], 128+128+64, [[128, 128, 256]
                                             , [128, 196, 256]])
        self.sa3 = PointNetSetAbstraction(npoint=None, radius=None,
                                             nsample=None, in_channel=512 + 3,
                                             mlp=[256, 512, 1024], group_all=
                                             True)

        self.mlp = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(),
            nn.Linear(512, out_channel))

    def forward(self, xyz):
        # Set Abstraction layers
        B,C,N = xyz.shape
        l0_points = xyz
        l0_xyz = xyz
        l1_xyz, l1_points = self.sa1(l0_xyz, l0_points)
        l2_xyz, l2_points = self.sa2(l1_xyz, l1_points)
        l3_xyz, l3_points = self.sa3(l2_xyz, l2_points)

        out_data = self.mlp(l3_points.squeeze(-1))
        return out_data
```

B.5. RPMG/utils/

B.5.1. rpmg.py

```python
import torch
import sys
import os
BASEPATH = os.path.dirname(__file__)
sys.path.append(BASEPATH)
import tools

def Rodrigues(w):
    '''
    axis angle -> rotation
    :param w: [b,3]
    :return: R: [b,3,3]
    '''
    w = w.unsqueeze(2).unsqueeze(3).repeat(1, 1, 3, 3)
    b = w.shape[0]
    theta = w.norm(dim=1)
    #print(theta[0])
    #theta = torch.where(t>math.pi/16, torch.Tensor([math.pi/16]).
                                              cuda(), t)
    wnorm = w / (w.norm(dim=1,keepdim=True)+0.001)
    #wnorm = torch.nn.functional.normalize(w,dim=1)
    I = torch.eye(3, device=w.get_device()).repeat(b, 1, 1)
    help1 = torch.zeros((b,1,3, 3), device=w.get_device())
    help2 = torch.zeros((b,1,3, 3), device=w.get_device())
    help3 = torch.zeros((b,1,3, 3), device=w.get_device())
    help1[:,:,1, 2] = -1
    help1[:,:,2, 1] = 1
    help2[:,:,0, 2] = 1
    help2[:,:,2, 0] = -1
    help3[:,:,0, 1] = -1
    help3[:,:,1, 0] = 1
    Jwnorm = (torch.cat([help1,help2,help3],1)*wnorm).sum(dim=1)

    return I + torch.sin(theta) * Jwnorm + (1 - torch.cos(theta)) *
                                              torch.bmm(Jwnorm, Jwnorm)

logger = 0
def logger_init(ll):
    global logger
    logger = ll
    print('logger init')

class RPMG(torch.autograd.Function):
    '''
    full version. See "simple_RPMG()" for a simplified version.
    Tips:
        1. Use "logger_init()" to initialize the logger, if you want to record some intermidiate variables by tensorboard.
        2. Use sum of L2/geodesic loss instead of mean, since our tau_converge is derivated without considering the scalar introduced by mean loss.
           See <ModelNet_PC> for an example.
        3. Pass "weight=\$YOUR_WEIGHT" instead of directly multiple the weight on rotation loss, if you want to reweight R loss and other losses.
           See <poselstm-pytorch> for an example.
    '''
    @staticmethod
    def forward(ctx, in_nd, tau, lam, rgt, iter, weight=1):
        proj_kind = in_nd.shape[1]
        if proj_kind == 6:
            r0 = tools.compute_rotation_matrix_from_ortho6d(in_nd)
        elif proj_kind == 9:
            r0 = tools.symmetric_orthogonalization(in_nd)
        elif proj_kind == 4:
            r0 = tools.compute_rotation_matrix_from_quaternion(in_nd)
        elif proj_kind == 10:
            r0 = tools.compute_rotation_matrix_from_10d(in_nd)
        else:
            raise NotImplementedError
        ctx.save_for_backward(in_nd, r0, torch.Tensor([tau,lam, iter, weight]), rgt)
        return r0

    @staticmethod
    def backward(ctx, grad_in):
        in_nd, r0, config,rgt,  = ctx.saved_tensors
        tau = config[0]
        lam = config[1]
        b = r0.shape[0]
        iter = config[2]
        weight = config[3]
        proj_kind = in_nd.shape[1]

        # use Riemannian optimization to get the next goal R
        if tau == -1:
            r_new = rgt
        else:
            # Eucliean gradient -> Riemannian gradient
            Jx = torch.zeros((b, 3, 3)).cuda()
            Jx[:, 2, 1] = 1
            Jx[:, 1, 2] = -1
            Jy = torch.zeros((b, 3, 3)).cuda()
            Jy[:, 0, 2] = 1
            Jy[:, 2, 0] = -1
            Jz = torch.zeros((b, 3, 3)).cuda()
            Jz[:, 0, 1] = -1
            Jz[:, 1, 0] = 1
            gx = (grad_in*torch.bmm(r0, Jx)).reshape(-1,9).sum(dim=1,keepdim=True)
            gy = (grad_in * torch.bmm(r0, Jy)).reshape(-1, 9).sum(dim=1,keepdim=True)
            gz = (grad_in * torch.bmm(r0, Jz)).reshape(-1, 9).sum(dim=1,keepdim=True)
            g = torch.cat([gx,gy,gz],1)

            # take one step
            delta_w = -tau * g

            # update R
            r_new = torch.bmm(r0, Rodrigues(delta_w))

            #this can help you to tune the tau if you don't use L2/geodesic loss.
            if iter % 100 == 0:
                logger.add_scalar('next_goal_angle_mean', delta_w.norm(dim=1).mean(), iter)
                logger.add_scalar('next_goal_angle_max', delta_w.norm(dim=1).max(), iter)
                R0_Rgt = tools.compute_geodesic_distance_from_two_matrices(r0, rgt)
                logger.add_scalar('r0_rgt_angle', R0_Rgt.mean(), iter)

        # inverse & project
        if proj_kind == 6:
            r_proj_1 = (r_new[:, :, 0] * in_nd[:, :3]).sum(dim=1, keepdim=True) * r_new[:, :, 0]
            r_proj_2 = (r_new[:, :, 0] * in_nd[:, 3:]).sum(dim=1, keepdim=True) * r_new[:, :, 0] \
                      + (r_new[:, :, 1] * in_nd[:, 3:]).sum(dim=1, keepdim=True) * r_new[:, :, 1]
            r_reg_1 = lam * (r_proj_1 - r_new[:, :, 0])
            r_reg_2 = lam * (r_proj_2 - r_new[:, :, 1])
            gradient_nd = torch.cat([in_nd[:, :3] - r_proj_1 + r_reg_1, in_nd[:, 3:] - r_proj_2 + r_reg_2], 1)
        elif proj_kind == 9:
            SVD_proj = tools.compute_SVD_nearest_Mnlsew(in_nd.reshape(-1,3,3), r_new)
            gradient_nd = in_nd - SVD_proj + lam * (SVD_proj - r_new.reshape(-1,9))
            R_proj_g = tools.symmetric_orthogonalization(SVD_proj)
            if iter % 100 == 0:
                logger.add_scalar('9d_reflection', (((R_proj_g-r_new).reshape(-1,9).abs().sum(dim=1))>5e-1).sum(), iter)
                logger.add_scalar('reg', (SVD_proj - r_new.reshape(-1, 9)).norm(dim=1).mean(), iter)
                logger.add_scalar('main', (in_nd - SVD_proj).norm(dim=1).mean(), iter)
        elif proj_kind == 4:
            q_1 = tools.compute_quaternions_from_rotation_matrices(r_new)
            q_2 = -q_1
            normalized_nd = tools.normalize_vector(in_nd)
            q_new = torch.where(
                (q_1 - normalized_nd).norm(dim=1, keepdim=True) < (q_2 - normalized_nd).norm(dim=1, keepdim=True),
                q_1, q_2)
            q_proj = (in_nd * q_new).sum(dim=1, keepdim=True) * q_new
            gradient_nd = in_nd - q_proj + lam * (q_proj - q_new)
        elif proj_kind == 10:
            qg = tools.compute_quaternions_from_rotation_matrices(r_new)
            new_x = tools.compute_nearest_10d(in_nd, qg)
            reg_A = torch.eye(4, device=qg.device)[None].repeat(qg.shape[0],1,1) - torch.bmm(qg.unsqueeze(-1), qg.unsqueeze(-2))
            reg_x = tools.convert_A_to_Avec(reg_A)
            gradient_nd = in_nd - new_x + lam * (new_x - reg_x)
            if iter % 100 == 0:
                logger.add_scalar('reg', (new_x - reg_x).norm(dim=1).mean(), iter)
                logger.add_scalar('main', (in_nd - new_x).norm(dim=1).mean(), iter)

        return gradient_nd * weight, None, None,None,None,None



class simple_RPMG(torch.autograd.Function):
    '''
    simplified version without tensorboard and r_gt.
    '''
    @staticmethod
    def forward(ctx, in_nd, tau, lam, weight=1):
        proj_kind = in_nd.shape[1]
        if proj_kind == 6:
            r0 = tools.compute_rotation_matrix_from_ortho6d(in_nd)
        elif proj_kind == 9:
            r0 = tools.symmetric_orthogonalization(in_nd)
        elif proj_kind == 4:
            r0 = tools.compute_rotation_matrix_from_quaternion(in_nd
                                                              )
        elif proj_kind == 10:
            r0 = tools.compute_rotation_matrix_from_10d(in_nd)
        else:
            raise NotImplementedError
        ctx.save_for_backward(in_nd, r0, torch.Tensor([tau,lam,
                                                        weight]))
        return r0

    @staticmethod
    def backward(ctx, grad_in):
        in_nd, r0, config,  = ctx.saved_tensors
        tau = config[0]
        lam = config[1]
        weight = config[2]
        b = r0.shape[0]
        proj_kind = in_nd.shape[1]

        # use Riemannian optimization to get the next goal R
        # Eucliean gradient -> Riemannian gradient
        Jx = torch.zeros((b, 3, 3)).cuda()
        Jx[:, 2, 1] = 1
        Jx[:, 1, 2] = -1
        Jy = torch.zeros((b, 3, 3)).cuda()
        Jy[:, 0, 2] = 1
        Jy[:, 2, 0] = -1
        Jz = torch.zeros((b, 3, 3)).cuda()
        Jz[:, 0, 1] = -1
        Jz[:, 1, 0] = 1
        gx = (grad_in*torch.bmm(r0, Jx)).reshape(-1,9).sum(dim=1,
                                                        keepdim=True)
        gy = (grad_in * torch.bmm(r0, Jy)).reshape(-1, 9).sum(dim=1,
                                                        keepdim=True)
        gz = (grad_in * torch.bmm(r0, Jz)).reshape(-1, 9).sum(dim=1,
                                                        keepdim=True)
        g = torch.cat([gx,gy,gz],1)

        # take one step
        delta_w = -tau * g

        # update R
        r_new = torch.bmm(r0, Rodrigues(delta_w))

        # inverse & project
        if proj_kind == 6:
            r_proj_1 = (r_new[:, :, 0] * in_nd[:, :3]).sum(dim=1,
                                                        keepdim=True) * r_new[:, :, 0]
            r_proj_2 = (r_new[:, :, 0] * in_nd[:, 3:]).sum(dim=1,
                                                        keepdim=True) * r_new[:, :, 0] \
                      + (r_new[:, :, 1] * in_nd[:, 3:]).sum(dim=1, keepdim=True) * r_new[:, :, 1]
            r_reg_1 = lam * (r_proj_1 - r_new[:, :, 0])
            r_reg_2 = lam * (r_proj_2 - r_new[:, :, 1])
            gradient_nd = torch.cat([in_nd[:, :3] - r_proj_1 + r_reg_1, in_nd[:, 3:] - r_proj_2 + r_reg_2], 1)
        elif proj_kind == 9:
            SVD_proj = tools.compute_SVD_nearest_Mnlsew(in_nd.reshape(-1,3,3), r_new)
            gradient_nd = in_nd - SVD_proj + lam * (SVD_proj - r_new.reshape(-1,9))
        elif proj_kind == 4:
            q_1 = tools.compute_quaternions_from_rotation_matrices(r_new)
            q_2 = -q_1
            normalized_nd = tools.normalize_vector(in_nd)
            q_new = torch.where(
                (q_1 - normalized_nd).norm(dim=1, keepdim=True) < (q_2 - normalized_nd).norm(dim=1, keepdim=True),
                q_1, q_2)
            q_proj = (in_nd * q_new).sum(dim=1, keepdim=True) * q_new
            gradient_nd = in_nd - q_proj + lam * (q_proj - q_new)
        elif proj_kind == 10:
            qg = tools.compute_quaternions_from_rotation_matrices(r_new)
            new_x = tools.compute_nearest_10d(in_nd, qg)
            reg_A = torch.eye(4, device=qg.device)[None].repeat(qg.shape[0],1,1) - torch.bmm(qg.unsqueeze(-1), qg.unsqueeze(-2))
            reg_x = tools.convert_A_to_Avec(reg_A)
            gradient_nd = in_nd - new_x + lam * (new_x - reg_x)

        return gradient_nd * weight, None, None,None,None,None
```

B.5.2. tools.py

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import numpy as np



#rotation5d batch*5
def normalize_5d_rotation( r5d):
    batch = r5d.shape[0]
    sin_cos = r5d[:,0:2] #batch*2
    sin_cos_mag = torch.max(torch.sqrt( sin_cos.pow(2).sum(1)), torch.autograd.Variable(torch.DoubleTensor([1e-8]).cuda()) ) #batch
    sin_cos_mag=sin_cos_mag.view(batch,1).expand(batch,2) #batch*2
    sin_cos = sin_cos/sin_cos_mag #batch*2

    axis = r5d[:,2:5] #batch*3
    axis_mag = torch.max(torch.sqrt( axis.pow(2).sum(1)), torch.autograd.Variable(torch.DoubleTensor([1e-8]).cuda()) ) #batch

    axis_mag=axis_mag.view(batch,1).expand(batch,3) #batch*3
    axis = axis/axis_mag #batch*3
    out_rotation = torch.cat((sin_cos, axis),1) #batch*5

    return out_rotation

#rotation5d batch*5
#out matrix batch*3*3
def rotation5d_to_matrix( r5d):

    batch = r5d.shape[0]
    sin = r5d[:,0].view(batch,1) #batch*1
    cos= r5d[:,1].view(batch,1) #batch*1

    x = r5d[:,2].view(batch,1) #batch*1
    y = r5d[:,3].view(batch,1) #batch*1
    z = r5d[:,4].view(batch,1) #batch*1

    row1 = torch.cat( (cos + x*x*(1-cos),  x*y*(1-cos)-z*sin, x*z*(1-cos)+y*sin ), 1) #batch*3
    row2 = torch.cat( (y*x*(1-cos)+z*sin,  cos+y*y*(1-cos),    y*z*(1-cos)-x*sin  ), 1) #batch*3
    row3 = torch.cat( (z*x*(1-cos)-y*sin,  z*y*(1-cos)+x*sin, cos+z*z*(1-cos)  ), 1) #batch*3

    matrix = torch.cat((row1.view(-1,1,3), row2.view(-1,1,3), row3.view(-1,1,3)), 1) #batch*3*3*seq_len
    matrix = matrix.view(batch, 3,3)
    return matrix

#T_poses num*3
#r_matrix batch*3*3
def compute_pose_from_rotation_matrix(T_pose, r_matrix):
    batch=r_matrix.shape[0]
    joint_num = T_pose.shape[0]
    r_matrices = r_matrix.view(batch,1, 3,3).expand(batch,joint_num,3,3).contiguous().view(batch*
                                    joint_num,3,3)
    src_poses = T_pose.view(1,joint_num,3,1).expand(batch,joint_num,
                                    3,1).contiguous().view(batch*
                                    joint_num,3,1)

    out_poses = torch.matmul(r_matrices, src_poses) #(batch*
                                    joint_num)*3*1

    return out_poses.view(batch, joint_num,3)

# batch*n
def normalize_vector( v):
    batch=v.shape[0]
    v_mag = torch.sqrt(v.pow(2).sum(1))# batch
    v_mag = torch.max(v_mag, torch.autograd.Variable(torch.
                                    FloatTensor([1e-8]).to(v.device)))
    v_mag = v_mag.view(batch,1).expand(batch,v.shape[1])
    v = v/v_mag
    return v

# u, v batch*n
def cross_product( u, v):
    batch = u.shape[0]
    #print (u.shape)
    #print (v.shape)
    i = u[:,1]*v[:,2] - u[:,2]*v[:,1]
    j = u[:,2]*v[:,0] - u[:,0]*v[:,2]
    k = u[:,0]*v[:,1] - u[:,1]*v[:,0]

    out = torch.cat((i.view(batch,1), j.view(batch,1), k.view(batch,
                                    1)),1)#batch*3

    return out


#poses batch*6
#poses
def compute_rotation_matrix_from_ortho6d(poses):
    x_raw = poses[:,0:3]#batch*3
    y_raw = poses[:,3:6]#batch*3

    x = normalize_vector(x_raw) #batch*3
    z = cross_product(x,y_raw) #batch*3
    z = normalize_vector(z)#batch*3
    y = cross_product(z,x)#batch*3

    x = x.view(-1,3,1)
    y = y.view(-1,3,1)
    z = z.view(-1,3,1)
    matrix = torch.cat((x,y,z), 2) #batch*3*3
    return matrix
```

```python

#u,a batch*3
#out batch*3
def proj_u_a(u,a):
    batch=u.shape[0]
    top = u[:,0]*a[:,0] + u[:,1]*a[:,1]+u[:,2]*a[:,2]
    bottom = u[:,0]*u[:,0] + u[:,1]*u[:,1]+u[:,2]*u[:,2]
    bottom = torch.max(torch.autograd.Variable(torch.zeros(batch).
                                  cuda())+1e-8, bottom)
    factor = (top/bottom).view(batch,1).expand(batch,3)
    out = factor* u
    return out

#matrices batch*3*3
def compute_rotation_matrix_from_matrix(matrices):
    b = matrices.shape[0]
    a1 = matrices[:,:,0]#batch*3
    a2 = matrices[:,:,1]
    a3 = matrices[:,:,2]

    u1 = a1
    u2 = a2 - proj_u_a(u1,a2)
    u3 = a3 - proj_u_a(u1,a3) - proj_u_a(u2,a3)

    e1 = normalize_vector(u1)
    e2 = normalize_vector(u2)
    e3 = normalize_vector(u3)

    rmat = torch.cat((e1.view(b, 3,1), e2.view(b,3,1),e3.view(b,3,1)
                                  ), 2)

    return rmat


#in batch*5
#out batch*6
def stereographic_unproject_old(a):

    s2 = torch.pow(a,2).sum(1) #batch
    unproj= 2*a/ (s2+1).view(-1,1).repeat(1,5) #batch*5
    w = (s2-1)/(s2+1) #batch
    out = torch.cat((unproj, w.view(-1,1)), 1) #batch*6

    return out

#in a batch*5, axis int
def stereographic_unproject(a, axis=None):
    """
  Inverse of stereographic projection: increases dimension by one.
  """
    batch=a.shape[0]
    if axis is None:
        axis = a.shape[1]
    s2 = torch.pow(a,2).sum(1) #batch
    ans = torch.autograd.Variable(torch.zeros(batch, a.shape[1]+1).cuda()) #batch*6
    unproj = 2*a/(s2+1).view(batch,1).repeat(1,a.shape[1]) #batch*5
    if(axis>0):
        ans[:,:axis] = unproj[:,:axis] #batch*(axis-0)
    ans[:,axis] = (s2-1)/(s2+1) #batch
    ans[:,axis+1:] = unproj[:,axis:]   #batch*(5-axis)    # Note that this is a no-op if the default option (last axis) is used
    return ans



#a batch*5
#out batch*3*3
def compute_rotation_matrix_from_ortho5d(a):
    batch = a.shape[0]
    proj_scale_np = np.array([np.sqrt(2)+1, np.sqrt(2)+1, np.sqrt(2)]) #3
    proj_scale = torch.autograd.Variable(torch.FloatTensor(proj_scale_np).cuda()).view(1,3).repeat(batch,1) #batch,3

    u = stereographic_unproject(a[:, 2:5] * proj_scale, axis=0)#batch*4
    norm = torch.sqrt(torch.pow(u[:,1:],2).sum(1)) #batch
    u = u/ norm.view(batch,1).repeat(1,u.shape[1]) #batch*4
    b = torch.cat((a[:,0:2], u),1)#batch*6
    matrix = compute_rotation_matrix_from_ortho6d(b)
    return matrix

#quaternion batch*4
def compute_rotation_matrix_from_quaternion( quaternion, n_flag=True):
    batch=quaternion.shape[0]
    if n_flag:
        quat = normalize_vector(quaternion)
    else:
        quat = quaternion
    qw = quat[...,0].view(batch, 1)
    qx = quat[...,1].view(batch, 1)
    qy = quat[...,2].view(batch, 1)
    qz = quat[...,3].view(batch, 1)

    # Unit quaternion rotation matrices computatation
    xx = qx*qx
    yy = qy*qy
    zz = qz*qz
    xy = qx*qy
    xz = qx*qz
    yz = qy*qz
    xw = qx*qw
    yw = qy*qw
    zw = qz*qw

    row0 = torch.cat((1-2*yy-2*zz, 2*xy - 2*zw, 2*xz + 2*yw), 1) #
                                  batch*3
    row1 = torch.cat((2*xy+ 2*zw,  1-2*xx-2*zz, 2*yz-2*xw  ), 1) #
                                  batch*3
    row2 = torch.cat((2*xz-2*yw,   2*yz+2*xw,   1-2*xx-2*yy), 1) #
                                  batch*3

    matrix = torch.cat((row0.view(batch, 1, 3), row1.view(batch,1,3)
                                  , row2.view(batch,1,3)),1) #batch*
                                  3*3

    return matrix

#axisAngle batch*4 angle, x,y,z
def compute_rotation_matrix_from_axisAngle(axisAngle):
    batch = axisAngle.shape[0]
    theta = axisAngle[:,0]
    #theta = torch.tanh(axisAngle[:,0])*np.pi #[-180, 180]
    sin = torch.sin(theta/2)
    axis = normalize_vector(axisAngle[:,1:4]) #batch*3
    qw = torch.cos(theta/2)
    qx = axis[:,0]*sin
    qy = axis[:,1]*sin
    qz = axis[:,2]*sin

    # Unit quaternion rotation matrices computatation
    xx = (qx*qx).view(batch,1)
    yy = (qy*qy).view(batch,1)
    zz = (qz*qz).view(batch,1)
    xy = (qx*qy).view(batch,1)
    xz = (qx*qz).view(batch,1)
    yz = (qy*qz).view(batch,1)
    xw = (qx*qw).view(batch,1)
    yw = (qy*qw).view(batch,1)
    zw = (qz*qw).view(batch,1)

    row0 = torch.cat((1-2*yy-2*zz, 2*xy - 2*zw, 2*xz + 2*yw), 1) #
                                  batch*3
    row1 = torch.cat((2*xy+ 2*zw,  1-2*xx-2*zz, 2*yz-2*xw  ), 1) #
                                  batch*3
    row2 = torch.cat((2*xz-2*yw,   2*yz+2*xw,   1-2*xx-2*yy), 1) #
                                  batch*3

    matrix = torch.cat((row0.view(batch, 1, 3), row1.view(batch,1,3), row2.view(batch,1,3)),1) #batch*3*3

    return matrix

#axisAngle batch*3 a,b,c
def compute_rotation_matrix_from_hopf( hopf):
    batch = hopf.shape[0]

    theta = (torch.tanh(hopf[:,0])+1.0)*np.pi/2.0 #[0, pi]
    phi   = (torch.tanh(hopf[:,1])+1.0)*np.pi     #[0,2pi)
    tao   = (torch.tanh(hopf[:,2])+1.0)*np.pi     #[0,2pi)

    qw = torch.cos(theta/2)*torch.cos(tao/2)
    qx = torch.cos(theta/2)*torch.sin(tao/2)
    qy = torch.sin(theta/2)*torch.cos(phi+tao/2)
    qz = torch.sin(theta/2)*torch.sin(phi+tao/2)

    # Unit quaternion rotation matrices computatation
    xx = (qx*qx).view(batch,1)
    yy = (qy*qy).view(batch,1)
    zz = (qz*qz).view(batch,1)
    xy = (qx*qy).view(batch,1)
    xz = (qx*qz).view(batch,1)
    yz = (qy*qz).view(batch,1)
    xw = (qx*qw).view(batch,1)
    yw = (qy*qw).view(batch,1)
    zw = (qz*qw).view(batch,1)

    row0 = torch.cat((1-2*yy-2*zz, 2*xy - 2*zw, 2*xz + 2*yw), 1) #batch*3
    row1 = torch.cat((2*xy+ 2*zw,  1-2*xx-2*zz, 2*yz-2*xw  ), 1) #batch*3
    row2 = torch.cat((2*xz-2*yw,   2*yz+2*xw,   1-2*xx-2*yy), 1) #batch*3

    matrix = torch.cat((row0.view(batch, 1, 3), row1.view(batch,1,3), row2.view(batch,1,3)),1) #batch*3*3

    return matrix


#euler batch*4
#output cuda batch*3*3 matrices in the rotation order of XZ'Y'' (intrinsic) or YZX (extrinsic)
def compute_rotation_matrix_from_euler(euler):
    batch=euler.shape[0]

    c1=torch.cos(euler[:,0]).view(batch,1)#batch*1
    s1=torch.sin(euler[:,0]).view(batch,1)#batch*1
    c2=torch.cos(euler[:,2]).view(batch,1)#batch*1
    s2=torch.sin(euler[:,2]).view(batch,1)#batch*1
    c3=torch.cos(euler[:,1]).view(batch,1)#batch*1
    s3=torch.sin(euler[:,1]).view(batch,1)#batch*1

    row1=torch.cat((c2*c3,          -s2,    c2*s3         ), 1).view
                              (-1,1,3) #batch*1*3
    row2=torch.cat((c1*s2*c3+s1*s3, c1*c2,  c1*s2*s3-s1*c3), 1).view
                              (-1,1,3) #batch*1*3
    row3=torch.cat((s1*s2*c3-c1*s3, s1*c2,  s1*s2*s3+c1*c3), 1).view
                              (-1,1,3) #batch*1*3

    matrix = torch.cat((row1, row2, row3), 1) #batch*3*3


    return matrix

#m batch*3*3
#out batch*4*4
def get_44_rotation_matrix_from_33_rotation_matrix(m):
    batch = m.shape[0]

    row4 = torch.autograd.Variable(torch.zeros(batch, 1,3).cuda())

    m43 = torch.cat((m, row4),1)#batch*4,3

    col4 = torch.autograd.Variable(torch.zeros(batch,4,1).cuda())
    col4[:,3,0]=col4[:,3,0]+1

    out=torch.cat((m43, col4), 2) #batch*4*4

    return out



#matrices batch*3*3
#both matrix are orthogonal rotation matrices
#out theta between 0 to 180 degree batch
def compute_geodesic_distance_from_two_matrices(m1, m2):
    batch=m1.shape[0]
    m = torch.bmm(m1, m2.transpose(1,2)) #batch*3*3

    cos = (  m[:,0,0] + m[:,1,1] + m[:,2,2] - 1 )/2
    cos = torch.min(cos, torch.autograd.Variable(torch.ones(batch).
                              cuda()) )
    cos = torch.max(cos, torch.autograd.Variable(torch.ones(batch).
                              cuda())*-1 )


    theta = torch.acos(cos)

    #theta = torch.min(theta, 2*np.pi - theta)


    return theta


#matrices batch*3*3
#both matrix are orthogonal rotation matrices
#out theta between 0 to pi batch
def compute_angle_from_r_matrices(m):

    batch=m.shape[0]

    cos = (  m[:,0,0] + m[:,1,1] + m[:,2,2] - 1 )/2
    cos = torch.min(cos, torch.autograd.Variable(torch.ones(batch).
                                    cuda()) )
    cos = torch.max(cos, torch.autograd.Variable(torch.ones(batch).
                                    cuda())*-1 )

    theta = torch.acos(cos)

    return theta

def get_sampled_rotation_matrices_by_quat(batch):
    #quat = torch.autograd.Variable(torch.rand(batch,4).cuda())
    quat = torch.autograd.Variable(torch.randn(batch, 4).cuda())
    matrix = compute_rotation_matrix_from_quaternion(quat)
    return matrix

def get_sampled_rotation_matrices_by_hpof(batch):

    theta = torch.autograd.Variable(torch.FloatTensor(np.random.
                                    uniform(0,1, batch)*np.pi).cuda())
                                     #[0, pi]
    phi   =  torch.autograd.Variable(torch.FloatTensor(np.random.
                                    uniform(0,2,batch)*np.pi).cuda())
                                         #[0,2pi)
    tao   = torch.autograd.Variable(torch.FloatTensor(np.random.
                                    uniform(0,2,batch)*np.pi).cuda())
                                         #[0,2pi)


    qw = torch.cos(theta/2)*torch.cos(tao/2)
    qx = torch.cos(theta/2)*torch.sin(tao/2)
    qy = torch.sin(theta/2)*torch.cos(phi+tao/2)
    qz = torch.sin(theta/2)*torch.sin(phi+tao/2)

    # Unit quaternion rotation matrices computatation
    xx = (qx*qx).view(batch,1)
    yy = (qy*qy).view(batch,1)
    zz = (qz*qz).view(batch,1)
    xy = (qx*qy).view(batch,1)
    xz = (qx*qz).view(batch,1)
    yz = (qy*qz).view(batch,1)
    xw = (qx*qw).view(batch,1)
    yw = (qy*qw).view(batch,1)
    zw = (qz*qw).view(batch,1)

    row0 = torch.cat((1-2*yy-2*zz, 2*xy - 2*zw, 2*xz + 2*yw), 1) #
                                batch*3
    row1 = torch.cat((2*xy+ 2*zw,  1-2*xx-2*zz, 2*yz-2*xw  ), 1) #
                                batch*3
    row2 = torch.cat((2*xz-2*yw,   2*yz+2*xw,   1-2*xx-2*yy), 1) #
                                batch*3

    matrix = torch.cat((row0.view(batch, 1, 3), row1.view(batch,1,3)
                                , row2.view(batch,1,3)),1) #batch*
                                3*3

    return matrix

#axisAngle batch*3*3s angle, x,y,z
def get_sampled_rotation_matrices_by_axisAngle( batch):

    theta = torch.autograd.Variable(torch.FloatTensor(np.random.
                                uniform(-1,1, batch)*np.pi).cuda()
                                ) #[0, pi] #[-180, 180]
    sin = torch.sin(theta)
    axis = torch.autograd.Variable(torch.randn(batch, 3).cuda())
    axis = normalize_vector(axis) #batch*3
    qw = torch.cos(theta)
    qx = axis[:,0]*sin
    qy = axis[:,1]*sin
    qz = axis[:,2]*sin

    # Unit quaternion rotation matrices computatation
    xx = (qx*qx).view(batch,1)
    yy = (qy*qy).view(batch,1)
    zz = (qz*qz).view(batch,1)
    xy = (qx*qy).view(batch,1)
    xz = (qx*qz).view(batch,1)
    yz = (qy*qz).view(batch,1)
    xw = (qx*qw).view(batch,1)
    yw = (qy*qw).view(batch,1)
    zw = (qz*qw).view(batch,1)

    row0 = torch.cat((1-2*yy-2*zz, 2*xy - 2*zw, 2*xz + 2*yw), 1) #
                                batch*3
    row1 = torch.cat((2*xy+ 2*zw,  1-2*xx-2*zz, 2*yz-2*xw  ), 1) #
                                batch*3
    row2 = torch.cat((2*xz-2*yw,   2*yz+2*xw,   1-2*xx-2*yy), 1) #
                                            batch*3

    matrix = torch.cat((row0.view(batch, 1, 3), row1.view(batch,1,3)
                                            , row2.view(batch,1,3)),1) #batch*
                                            3*3

    return matrix


#input batch*4*4 or batch*3*3
#output torch batch*3 x, y, z in radiant
#the rotation is in the sequence of x,y,z
def compute_euler_angles_from_rotation_matrices(rotation_matrices):
    batch=rotation_matrices.shape[0]
    R=rotation_matrices
    sy = torch.sqrt(R[:,0,0]*R[:,0,0]+R[:,1,0]*R[:,1,0])
    singular= sy<1e-6
    singular=singular.float()

    x=torch.atan2(R[:,2,1], R[:,2,2])
    y=torch.atan2(-R[:,2,0], sy)
    z=torch.atan2(R[:,1,0],R[:,0,0])

    xs=torch.atan2(-R[:,1,2], R[:,1,1])
    ys=torch.atan2(-R[:,2,0], sy)
    zs=R[:,1,0]*0

    out_euler=torch.autograd.Variable(torch.zeros(batch,3).cuda())
    out_euler[:,0]=x*(1-singular)+xs*singular
    out_euler[:,1]=y*(1-singular)+ys*singular
    out_euler[:,2]=z*(1-singular)+zs*singular

    return out_euler

#input batch*4
#output batch*4
def compute_quaternions_from_axisAngles(self, axisAngles):
    w = torch.cos(axisAngles[:,0]/2)
    sin = torch.sin(axisAngles[:,0]/2)
    x = sin*axisAngles[:,1]
    y = sin*axisAngles[:,2]
    z = sin*axisAngles[:,3]

    quat = torch.cat((w.view(-1,1), x.view(-1,1), y.view(-1,1), z.
                                            view(-1,1)), 1)

    return quat

#quaternions batch*4,
#matrices batch*4*4 or batch*3*3
```

```
def compute_quaternions_from_rotation_matrices(matrices):
    batch=matrices.shape[0]

    w=torch.sqrt(torch.max(1.0 + matrices[:,0,0] + matrices[:,1,1] +
                                 matrices[:,2,2], torch.zeros(1).
                                cuda())) / 2.0
    w = torch.max (w , torch.autograd.Variable(torch.zeros(batch).
                                cuda())+1e-8) #batch
    w4 = 4.0 * w
    x= (matrices[:,2,1] - matrices[:,1,2]) / w4
    y= (matrices[:,0,2] - matrices[:,2,0]) / w4
    z= (matrices[:,1,0] - matrices[:,0,1]) / w4
    quats = torch.cat( (w.view(batch,1), x.view(batch, 1),y.view(
                                batch, 1), z.view(batch, 1) ), 1
                                 )
    quats = normalize_vector(quats)
    return quats


def compute_v_wave(u, r_new):
    u_star = r_new[:, :, 0]
    u_out = normalize_vector(u)
    u_2 = normalize_vector(cross_product(u_out, u_star))
    real_angle = torch.acos(torch.clamp((u_out * u_star).sum(dim=1,
                                keepdim=True), -1, 1))
    ro = compute_rotation_matrix_from_axisAngle(torch.cat([
                                real_angle / 2, u_2], 1))
    v_new = torch.bmm(r_new.transpose(1, 2), ro)[:, 1, :]
    return v_new

def symmetric_orthogonalization(x):
  """Maps 9D input vectors onto SO(3) via symmetric
                                orthogonalization.
  x: should have size [batch_size, 9]
  Output has size [batch_size, 3, 3], where each inner 3x3 matrix is
                                 in SO(3).
  """
  m = x.view(-1, 3, 3)
  d = m.device
  u, s, v = torch.svd(m.cpu())
  u, v = u.to(d), v.to(d)
  vt = torch.transpose(v, 1, 2)
  det = torch.det(torch.bmm(u, vt))
  det = det.view(-1, 1, 1)
  vt = torch.cat((vt[:, :2, :], vt[:, -1:, :] * det), 1)
  r = torch.bmm(u, vt)
  return r

def compute_SVD_nearest_Mnlsew(R, Rg):
    '''
    solve the minimum problem       Find X to minimizing L2(R - S*
                                   Rg) while S is a symmetry matrix
    :param R: Network output Rotation matrix [b, 3, 3]
    :param Rg: next_goal Rotation matrix [b,3,3]
    :return: M
    '''
    S = (torch.bmm(R, Rg.transpose(2,1))+torch.bmm(Rg,R.transpose(2,
                                   1)))/2
    M = torch.bmm(S, Rg)
    return M.reshape(-1,9)

def convert_Avec_to_A(A_vec):
    """ Convert BxM tensor to BxNxN symmetric matrices """
    """ M = N*(N+1)/2"""
    if A_vec.dim() < 2:
        A_vec = A_vec.unsqueeze(dim=0)

    if A_vec.shape[1] == 10:
        A_dim = 4
    elif A_vec.shape[1] == 55:
        A_dim = 10
    else:
        raise ValueError("Arbitrary A_vec not yet implemented")

    idx = torch.triu_indices(A_dim, A_dim)
    A = A_vec.new_zeros((A_vec.shape[0], A_dim, A_dim))
    A[:, idx[0], idx[1]] = A_vec
    A[:, idx[1], idx[0]] = A_vec
    # return A.squeeze()
    return A

def convert_A_to_Avec(A):
    """ Convert BxNxN symmetric matrices to BxM tensor"""
    """ M = N*(N+1)/2"""
    idx = torch.triu_indices(4, 4)
    A_vec = A[:, idx[0], idx[1]]
    return A_vec

def compute_rotation_matrix_from_10d(x):
    A = convert_Avec_to_A(x)
    d = A.device
    _, evs = torch.symeig(A.cpu(), eigenvectors=True)
    evs = evs.to(d)
    q = evs[:,:,0]
    return compute_rotation_matrix_from_quaternion(q,n_flag=False)


#x: [B, 10] raw output of network
#qg: [B, 4] updated quaternion
def compute_nearest_10d(x, qg, prev_eigenval=None):
    # [4,4]*[4,1] -> [4,10]*[10,1]
    d = qg.device
    b = qg.shape[0]
    assert len(qg.shape) == 2
    X_matrix = torch.zeros((b,4,10),device=d)
    Id = torch.eye(10,device=d)[None,...].repeat(b,1,1)
    Ze = torch.zeros((b,4,4),device=d)
    X_matrix[:, 0,0:4] = qg
    X_matrix[:, 1,[1,4,5,6]] = qg
    X_matrix[:, 2,[2,5,7,8]] = qg
    X_matrix[:, 3,[3,6,8,9]] = qg

    #[[I, X_m^T],[X_m, 0]]
    KKT_l = torch.cat([Id, X_matrix], dim=1)
    KKT_r = torch.cat([X_matrix.transpose(-1,-2), Ze], dim=1)
    KKT = torch.cat([KKT_l, KKT_r], dim=2)
    KKT_part = torch.inverse(KKT)[:, :10, -4:]

    qgs = qg.unsqueeze(-1)
    A = convert_Avec_to_A(x)
    Aqs = torch.bmm(A, qgs)
    if prev_eigenval is None:
        KKT_M = torch.bmm(KKT_part.transpose(-1,-2), KKT_part)
        eigenval = (torch.bmm(torch.bmm(qgs.transpose(-1,-2), KKT_M)
                                  ,Aqs)+torch.bmm(torch.bmm(Aqs.
                                  transpose(-1,-2), KKT_M), qgs))/(2
                                  *torch.bmm(torch.bmm(qgs.transpose
                                  (-1,-2), KKT_M), qgs))
    else:
        eigenval = prev_eigenval
    new_M = torch.bmm(KKT_part, eigenval*qgs-Aqs)
    new_x = new_M.squeeze()+x
    return new_x
```