ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Neural Architecture Search on Lie Groups for Skeleton-based Action Recognition

**Semester Project**

**Samuele Serafino**

**November 30, 2020**

**Supervisors:** Dr. Zhiwu Huang, Dr. Suryansh Kumar, Prof. Dr. Luc Van Gool

# Abstract

In recent years, neural architecture search (NAS) has drawn increasing attention in the field of computer vision. While state-of-the-art methods apply an architecture search on the Euclidean manifold, this project adapts the NAS problem to Lie Groups. The backbone of our implementation is the rotation cell which operates on Lie Group data and preserves the manifold properties. In order to instantiate a rotation cell, we introduce a new search space, containing several candidate operations to optimally process the input data. The cell is then built upon a mixture of the candidate operations. Instead of optimizing over a discrete and non-differentiable search space, our NAS model relies on the continuous relaxation of the search space. This enables us to apply standard optimization techniques and obtain a suitable network architecture. Experimental results show that our new model performs better at skeletal-based human action recognition than the state-of-the-art classification network LieNet. Thanks to the simultaneous architecture and model weights optimization, the training duration also is reduced.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

3D human action recognition has become more popular in recent years. Especially for autonomous robotic systems that interact with environments where humans are located, it is crucial to detect human activities reliably. To tackle this problem, [1], [21] and [22] represent human movements by skeletal data which resides on a Lie group, i.e., a non-Euclidean manifold. Action recognition models proposed in [1], [21] and [22] are rather slow and cannot be used for real-time applications. Furthermore, the proposed models are limited to linear learning schemes and apply two-step systems that typically perform worse than end-to-end learning schemes. Therefore, deep learning models are more suitable thanks to their ability to perform non-linear computations. The first neural network architecture that incorporates the Lie group structure is LieNet [6]. This classifier has achieved state-of-the-art performances for some 3D human action recognition benchmarks, one of which is the G3D Gaming data set. Generally, designing such deep networks requires a lot of time, effort, and domain expertise. For this reason, researchers have started developing algorithms to automate the process of neural architecture design [23] [24] [9] [15] [10] [19]. Despite the great potential of these neural architecture search (NAS) algorithms, they are limited to handle architectures with Euclidean operations and representations. In this project, we want to adapt the NAS process to Lie Groups. For this purpose, we follow the SPDNetNAS [18] implementation; the lately released NAS algorithm for SPD-manifold networks, i.e., models that deal with symmetric positive definite matrices as input data. The motivation of this project is to run an automated neural architecture search that results in a new model with a performance at least as high as LieNet's one. Our NAS problem requires an adaptation of the computation cell definition as well as a new search space containing Lie group candidate operations. Same as [10] and [18], we model the basic architecture cell with a specific directed acyclic graph (DAG). In our case, each node is a Lie group representation, and each edge corresponds to a Lie group operation. Our solution relies on the continuous relaxation of the defined search space, and therefore, we can employ a gradient descent approach to find a suitable architecture. The new model is evaluated on the G3D gaming data set and outperforms LieNet in terms of test accuracy. Our works makes the following contributions:

- We extend LieNet by a batch normalization and an average pooling layer. The input data to these layers consists of tuples of rotation matrices. With the new operations, we can construct a better generalizing model and hence leverage the performance. In addition, we have more candidate operations for the architecture search.
- We introduce a NAS problem adapted to Lie group data. The resulting model has a higher accuracy than LieNet on the G3D gaming data set, and in future it can be

used for different data sets as well.

We start by presenting the background and relevant work that we build up on in this work. Afterwards, we introduce new operations to extend the current framework and define the neural architecture search problem of Lie Groups. Then, we describe the approach that leads us to the solution of the specified problem. In the last part, the results are presented and discussed.

# Chapter 2

# Background

This chapter provides the necessary background on Lie groups which is used in the remainder of the thesis. Furthermore, we present the rotation layers used in LieNet which are analogous to standard convolutional neural network layers (e.g. convolution and pooling). In the last section, we introduce the concept of neural architecture search along with the implementation of DARTS.

## 2.1 Lie group and SO(3)

A Lie group is a differentiable manifold with a group structure such that the group multiplication and inverse-taking functions are smooth functions.
The 3D rotation group SO(3) is the group of all rotations around the origin of the euclidean space $\mathbb{R}^3$. A matrix $B$ is called a rotation matrix if it holds that $B^\top B = \mathbb{1}$ and $det(B) = 1$. Since SO(3) is a differentiable manifold and the multiplication of rotation matrices is a smooth operation, the 3D rotation group is also a Lie group.

## 2.2 Operations on the SO(3) manifold

For simplicity, we denote the space of 3 x 3 rotation matrices as $SO_3$ and the tangent space at some point $R_0$ as $T_{R_0}SO_3$, where $R_0 \in SO_3$. The tangent space is a euclidean vector space, meaning that operations defined on the euclidean space are valid in the tangent space. In order to operate on the $SO_3$ manifold, notions like origin, mean, identity or distance need to be adapted to the non-euclidean space. We shortly provide an overview of some operations that are used for this project.

### 2.2.1 Distance metric

Let $X$ and $Y$ be two points on $SO_3$. The closed form expression for the distance between $X$ and $Y$ is

$$d_R(X, Y) = \| \ logm(X^\top Y) \ \|_F \tag{2.1}$$

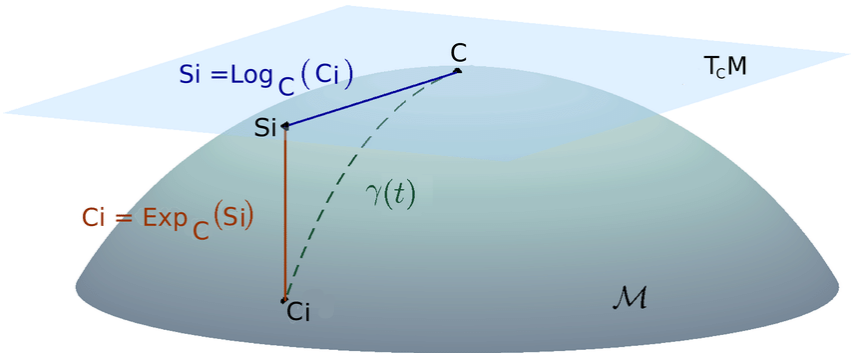


Figure 2.1: Visualization of a manifold $\mathcal{M}$ and its tangent space $T_C M$ at point C. [16]

where $logm(.)$ is the matrix logarithm and $\| \, . \, \|_F$ is the Frobenius norm. Note that $SO_3$ is a unit sphere, meaning that the shortest path from $X$ to $Y$ on the manifold is not simply a straight line, but a curve.

### 2.2.2 Exponential and Logarithmic map

Two important operations are the exponential and logarithmic map. The logarithmic map is a mapping from $SO_3$ to its tangent space, whereas the exponential map projects points from the tangent space back to the manifold as illustrated in Figure 2.1. Formally,

$$log_P(X) = logm(XP^\top) \text{ with } P, X \in SO_3, \tag{2.2}$$

$$exp_P(Z) = expm(ZP^\top) \text{ with } Z \in T_P SO_3 \tag{2.3}$$

where logm(.) and expm(.) are the matrix logarithm and exponential, respectively.

### 2.2.3 Identity element

Every group $G$ contains an identity element $\mathbb{1}$ such that $A \cdot \mathbb{1} = A$ and $\mathbb{1} \cdot A = A$, where $A \in G$. The identity element of the Lie Group $SO_3$ is the 3x3 identity matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The tangent space at identity is called Lie algebra. We make use of the Lie algebra when defining neural network layers operating on Lie Group data.

### 2.2.4 Definition of mean on SO(3)

Given a set of N 3x3 rotation matrices $\{R_i\}_{i=1}^N$, we can compute the Riemannian barycenter ($\mathfrak{G}$) as

$$\mathfrak{G} = \underset{R_\mu \in SO_3}{\arg\min} \sum_{i=1}^{N} d_R^2(R_i, R_\mu). \tag{2.4}$$

This expression is also referred as Fréchet mean [12] [2]. It can be extended to calculate

the weighted Riemannian mean or weighted Fréchet mean

$$\mathfrak{G} = \underset{R_\mu \in SO_3}{\arg\min} \sum_{i=1}^{N} w_i d_R^2(R_i, R_\mu), \text{ where } w_i \geq 0 \text{ and } \sum_{i=1}^{N} w_i = 1. \tag{2.5}$$

One method to approximate equation 2.2.4 is the Karcher flow [7]. The idea of this algorithm is to map the rotation matrices from the manifold to the tangent space, compute the euclidean mean on the tangent space and map it back to the manifold. It is a computationally efficient method and the arising imprecisions are negligible. Algorithm 1 is taken from [17].

**Algorithm 1** Karcher flow [7] to compute the Riemannian mean of N rotation matrices

**Require:** data points $\{P_i\}_{i \leq N}$, iterations K, step $\alpha$

1: $\mathfrak{G} \leftarrow \sum_{i \leq N} P_i$
2: **for** $k \leq K$ **do**
3: $\quad G \leftarrow \frac{1}{N} \sum_{i \leq N} log_{\mathfrak{G}}(P_i)$
4: $\quad \mathfrak{G} \leftarrow exp_{\mathfrak{G}}(\alpha G)$
5: **end for**
6: **return** $\mathfrak{G}$

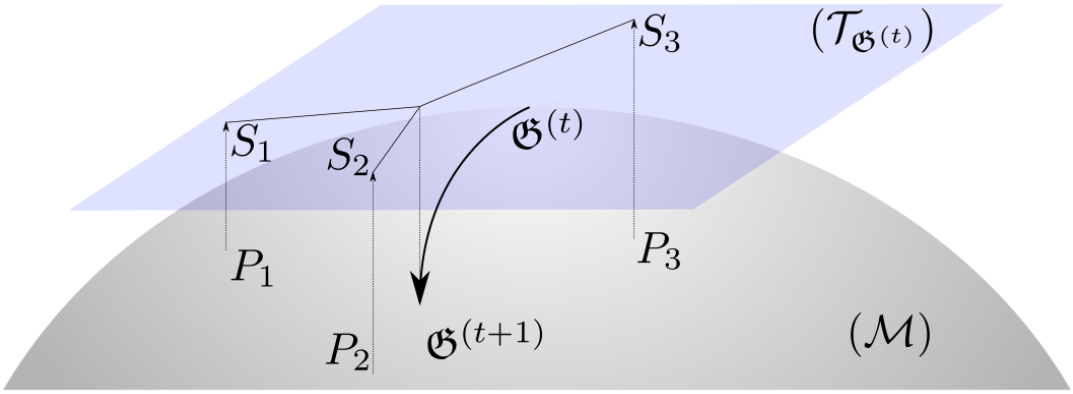


Figure 2.2: One iteration of the Karcher flow algorithm. The matrices $P_i$ are mapped to the tangent space at $\mathfrak{G}(t)$. Then, the euclidean mean is computed and mapped back to the manifold, yielding $\mathfrak{G}(t+1)$. [4]

## 2.3 LieNet

LieNet is the state-of-the-art deep learning model that incorporates Lie Group structured data. The goal of this project is to further improve the model. In the current framework, there are three different rotation layers which are described in the next sections. In addition, we expand the set of layers by two more operations such that we have more variability in the architecture search phase. This section's content is taken from [6].

Figure 2.3: A colored image with its associated depth image and extracted skeleton. [3]

### 2.3.1 Lie group representation of Skeletal Data

Let $S = (V, E)$ be a body skeleton, where $V = \{v_1, ..., v_N\}$ denotes the set of body joints, and $E = \{\mathbf{e}_1, ..., \mathbf{e}_M\}$ indicates the set of edges, i.e. the body bones. One can describe the coordinate system of $\mathbf{e}_m$ in the local coordinate system of $\mathbf{e}_n$ by rotating and translating the former one. Since the translational component is superfluous, the relation between two body parts is represented by a rotation matrix only. This means, we can compute a rotation matrix $\mathbf{R}_{m,n}$ from $\mathbf{e}_m$ to the coordinate system of $\mathbf{e}_n$. Analogously, $\mathbf{R}_{n,m}$ goes in the opposite direction. To fully encode the relative geometry between $\mathbf{e}_m$ and $\mathbf{e}_n$, both $\mathbf{R}_{m,n}$ and $\mathbf{R}_{n,m}$ are used. In this way, a skeleton S at time t is represented by the tuple $(\mathbf{R}_{1,2}(t), \mathbf{R}_{2,1}(t), ..., \mathbf{R}_{M-1,M}(t), \mathbf{R}_{M,M-1}(t))$, where $M$ is the number of body parts, and the number of rotation matrices at time t is $2M(M-1)$.

### 2.3.2 RotMap Layer

Similar to classical convolutional neural networks, LieNet exhibits convolutional-like layers which are called Rotation mapping layers or RotMap for short. RotMap layers transfrom the input skeleton to a new one in a way such that the new skeleton is better aligned and hence more suitable for classification. Formally, the application of a RotMap layer is defined as

$$\begin{aligned} & f_r^{(k)}((\mathbf{R}_1^{k-1}, \mathbf{R}_2^{k-1}, ..., \mathbf{R}_{\hat{M}}^{k-1}); \mathbf{W}_1^k, \mathbf{W}_2^k, ..., \mathbf{W}_{\hat{M}}^k) \\ & = (\mathbf{W}_1^k \mathbf{R}_1^{k-1}, \mathbf{W}_2^k \mathbf{R}_2^{k-1}, ..., \mathbf{W}_{\hat{M}}^k \mathbf{R}_{\hat{M}}^{k-1}) \\ & = (\mathbf{R}_1^k, \mathbf{R}_2^k, ..., \mathbf{R}_{\hat{M}}^k), \end{aligned} \tag{2.6}$$

where $\hat{M} = 2M(M-1)$ ($M$ is the number of body parts in one skeleton), $(\mathbf{R}_1^{k-1}, \mathbf{R}_2^{k-1}, ..., \mathbf{R}_{\hat{M}}^{k-1}) \in SO_3 \times SO_3 ... \times SO_3$ is the input Lie group feature for one skeleton in the k-th layer, $\mathbf{W}_i^k \in \mathbb{R}^{3\times 3}$ is the transformation matrix (connections weights), and $(\mathbf{R}_1^k, \mathbf{R}_2^k, ..., \mathbf{R}_{\hat{M}}^k)$ is the resulting Lie group representation. In order to make sure that the features remain on the Lie group $SO_3 \times SO_3 \times ... \times SO_3$ after applying one or more RotMap layers, the transformation matrices $(\mathbf{W}_1^k, \mathbf{W}_2^k, ..., \mathbf{W}_{\hat{M}}^k)$ are required to be rotation matrices. The justification of this requirement is that $SO_3$ is closed under matrix multiplication.

### 2.3.3 RotPooling Layer

As well-known in convolutional neural networks, it is useful and often even necessary to reduce the dimensionality of the input data for computational feasibility reasons. The

pooling operation down-samples a feature map by summarizing local statistics in neighborhoods within the feature map. The RotPooling layer adapts this concept to Lie group based data. There are two different notions for neighborhood in the LieNet pipeline. The first one is on the spatial level. Here, the Lie group features are pooled on each pair of basic bones $\mathbf{e}_m$, $\mathbf{e}_n$ in the i-th frame, which is represented by the two rotation matrices $\mathbf{R}_{m,n}^{k-1,i}$, $\mathbf{R}_{n,m}^{k-1,i}$ as explained before. The function of the max pooling is given by

$$\begin{aligned} & f_p^{(k)}(\{\mathbf{R}_{m,n}^{k-1,i}, \mathbf{R}_{n,m}^{k-1,i}\}) = \max(\{\mathbf{R}_{m,n}^{k-1,i}, \mathbf{R}_{n,m}^{k-1,i}\}) \\ & = \begin{cases} \mathbf{R}_{m,n}^{k-1,i} & \text{, if } \theta(\mathbf{R}_{m,n}^{k-1,i}) > \theta(\mathbf{R}_{n,m}^{k-1,i}) \\ \mathbf{R}_{n,m}^{k-1,i} & \text{, otherwise,} \end{cases} \end{aligned} \tag{2.7}$$

where $\theta(\,\cdot\,)$ is the Euler angle representation which is defined as

$$\theta(\mathbf{R}_{n,m}) = \arccos\left(\frac{trace(\mathbf{R}_{n,m}) - 1}{2}\right). \tag{2.8}$$

The other pooling variant is on the temporal level. The purpose of the temporal pooling is to obtain more compact representations for a motion sequence. Considering that a video sequence often contains many frames, temporal pooling reduces the model complexity as well. Formally, the second type of pooling is defined as

$$\begin{aligned} & f_p^{(k)}\left(\{(\mathbf{R}_{1,2}^{k-1,1}...\mathbf{R}_{M-1,M}^{k-1,1}), ..., (\mathbf{R}_{1,2}^{k-1,p}...\mathbf{R}_{M-1,M}^{k-1,p})\}\right) \\ & = \left(\max\left(\{\mathbf{R}_{1,2}^{k-1,1}, ..., \mathbf{R}_{1,2}^{k-1,p}\}\right), ..., \max\left(\{\mathbf{R}_{M-1,M}^{k-1,1}, ..., \mathbf{R}_{M-1,M}^{k-1,p}\}\right)\right), \end{aligned} \tag{2.9}$$

where M is the number of body parts in one skeleton, p is the number of video frames, and the function $\max(\,\cdot\,)$ is defined in equation 2.7.

### 2.3.4 LogMap Layer

Due to the non-euclidean nature of the skeletal data, classifying curves on the Lie group $SO_3 \times ... \times SO_3$ is a challenging task. In this regard, the LogMap layer is designed to flatten the Lie group $SO_3 \times ... \times SO_3$ to its Lie algebra $so_3 \times ... \times so_3$. Using equation 2.2, this layer is defined as

$$\begin{aligned} & f_l^{(k)}\left((\mathbf{R}_1^{k-1}, \mathbf{R}_2^{k-1}, ..., \mathbf{R}_{\hat{M}}^{k-1})\right) \\ & = \left(log_{\mathbb{1}}(\mathbf{R}_1^{k-1}), log_{\mathbb{1}}(\mathbf{R}_2^{k-1}), ..., log_{\mathbb{1}}(\mathbf{R}_{\hat{M}}^{k-1})\right) \\ & = \left(logm(\mathbf{R}_1^{k-1}), logm(\mathbf{R}_2^{k-1}), ..., logm(\mathbf{R}_{\hat{M}}^{k-1})\right). \end{aligned} \tag{2.10}$$

One typical approach to calculate the matrix logarithm is via the eigenvalue decomposition $logm(\mathbf{R}) = \mathbf{U}logm(\mathbf{\Sigma})\mathbf{U}^\top$, where $\mathbf{R} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}$ and $logm(\mathbf{\Sigma})$ is the diagonal matrix of the eigenvalue logarithms. One issue this method encounters is that the eigenvalues of rotation matrices are complex, except for the case $\mathbf{R} = \pm\mathbb{1}$ (positive or negative identity). That means, the LogMap layer would create complex values which is not desired for classification. Furthermore, the matrix gradient calculation is computationally expensive and hence consume too much time. One way to bypass these issues is to exploit the

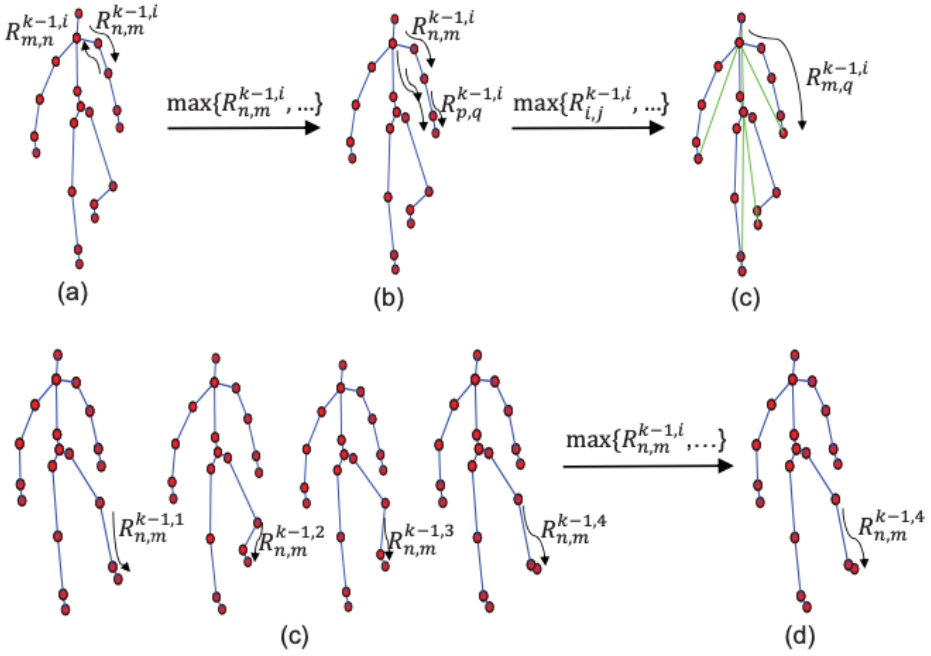


Figure 2.4: Illustration of the spatial pooling (a) → (b) → (c) and the temporal pooling (c) → (d) type [6]

relationship between the logarithm map and the axis-angle representation:

$$logm(\mathbf{R}) = \begin{cases} 0 & \text{, if } \theta(\mathbf{R}) \\ \frac{\theta(\mathbf{R})}{2\sin(\theta(\mathbf{R}))}(\mathbf{R} - \mathbf{R}^\top) & \text{, otherwise,} \end{cases} \tag{2.11}$$

where $\theta(\mathbf{R})$ is the angle of $\mathbf{R}$ as defined in equation 2.8. With this equation, the corresponding matrix gradient can be easily derived by traditional element-wise matrix calculation.

### 2.3.5 Output Layers

After performing the LogMap layers, the outputs can be transformed into vector form and concatenated directly frame by frame within one sequence due to their Euclidean nature. There is the possibility of adding any regular layer such as rectified linear unit (ReLU) layers and regular fully connected (FC) layers. For classification, a common softmax layer is employed as final output layer. An overview of LieNet is found in figure 2.5

## 2.4 Neural Architecture Search

This section's content is taken from [10] and [17].
Finding state-of-the-art neural network architectures requires a huge effort of human experts. For this reason, the demand for an automated architecture search has increased substantially. Also the large availability and access to high performance computing systems has encouraged researchers to develop new algorithmic solutions. Despite their

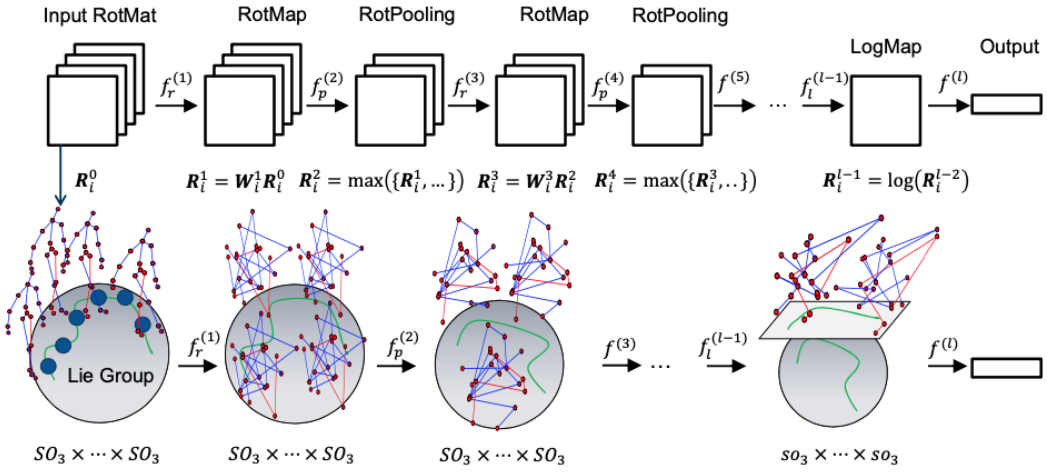


Figure 2.5: Conceptual illustration of the LieNet architecture. [6]

remarkable performance, earlier architecture search algorithms require thousands of GPU days for training. Most of recent approaches involve training a SuperNet which incorporates many candidate sub-networks.
In this project we use one of the most promising SuperNet strategy called Differentiable Architecture Search (DARTS). Instead of searching over a discrete set of candidate operations, DARTS uses a continuous search space, so that the architecture can be optimized via gradient descent. The advantage of this method is the achievement of competitive performance with the state of the art using orders of magnitude less computation resources.

DARTS represents the computation procedure for an architecture as a directed acyclic graph. A cell is a directed acyclic graph consisting of an ordered sequence of N nodes. Each node $x^{(i)}$ is a latent representation (e.g. a feature map in convolutional networks) and each directed edge $(i, j)$ is associated with some operation $o^{(i,j)}$ that transforms $x^{(i)}$. Cells are assumed to have two input nodes and a single output node. The output of the cell is obtained by applying a reduction operation (e.g. concatenation) to all the intermediate nodes. Each intermediate node is derived from all its predecessors.

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^i) \tag{2.12}$$

For a set of candidate operations $\mathcal{O}$, each operation represents some function $o(\cdot)$ to be applied to $x^{(i)}$. The mixing operation between a pair of nodes $(i, j)$ is a linear combination of the candidate operations, defined as

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} exp(\alpha_{o'}^{(i,j)})} o(x), \tag{2.13}$$

where $\alpha^{(i,j)}$ is the parametrization vector of the operation weights on edge $(i, j)$. The weights for operations on the edges, i.e $\alpha$'s are updated using the validation set while the weights of the architecture $w$'s are updated using a training set. The architecture and operation mixing weights are updated using a bi-level optimization as shown in algorithm

2.

**Algorithm 2** DARTS bi-level optimization algorithm [7]

Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge $(i, j)$

**while** not converged **do**

Update architecture $\alpha$ by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$

($\xi = 0$ if using first-order approximation)

Update weights w by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$

**end while**

Derive the final architecture based on the learned $\alpha$.

# Chapter 3

# LieNet extension

In this chapter, we extend LieNet by a batch normalization layer which is our first contribution in this project. Afterwards, we define the NAS problem that is solved in chapter 4.

## 3.1 BatchNormalize Layer

Training a deep learning model is difficult because the distribution of each layer's input data changes during the training phase. This can slow down the training but it can also lead to very high gradients. One widely used remedy for gradient explosion is the batch normalization. The idea of batch normalization is to standardize the data before passing it to a layer, resulting in a shorter training time of the overall network. The classical batch normalization algorithm for euclidean data is shown in figure 3.1.

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 3.1: Batch Normalization algorithm for euclidean data [5]

We want to adapt this layer to the Lie group data. Algorithms 3 and 4 show the normalization process on a Lie group for training and testing, respectively.
To summarize the concept, we first adjust the batch mean to be $\mathbb{1}$ (identity matrix), then we scale the data by $s$ and at the end the batch mean is shifted from $\mathbb{1}$ to $g$. Again, *logm* and *expm* refer to the matrix logarithm and exponential, respectively.

**Algorithm 3** Training step of normalization on a Lie group [5]

**Input:** A batch of samples $S = \{X_i\}_{i=1}^N$; bias $g \in SO_3$; running mean $M$; scale factor $s > 0$.
**Output:** Updated running mean $M$.

1: Compute batch mean $M_b$ of $\{X_i\}_{i=1}^N$ (using Karcher flow);
2: Update running mean $M$ by $M_b$ (using Karcher flow);
3: $X_i \leftarrow M_b^{-1} X_i$;
4: $X_i \leftarrow expm(s \cdot logm(X_i))$;
5: $X_i \leftarrow gX_i$.

**Algorithm 4** Testing step of normalization on a Lie group [5]

**Input:** A batch of samples $S = \{X_i\}_{i=1}^N$; bias $g \in SO_3$; learned running mean $M$; scale factor $s > 0$.

1: $X_i \leftarrow M^{-1} X_i$;
2: $X_i \leftarrow expm(s \cdot logm(X_i))$;
3: $X_i \leftarrow gX_i$;

At this point, we add the batch normalization layer to the original LieNet implementation. The resulting model is called LieNetBN and is illustrated in figure 5.5.

## 3.2 Problem Definition

The scope of this project is to extend DARTS in a way such that we can run an architecture on Lie group data. [17] has adapted the DARTS framework to SPD valued data, i.e. to the manifold of symmetric positive definite matrices. We try to remain close to the SPDNetNAS [17] implementation. In a first step, the computation cell has to be re-defined. It is important for the cell to operate on the $SO_3$ manifold, meaning that the output data of a cell lies on the Lie group. There are two types of cells. The first one is called a normal cell and outputs a feature map of the same dimensions as the input, whereas the second cell, also called reduction cell, reduces the dimensionality of the spatial or temporal domain. In the second step, we introduce a new search space in order to generate valid computation cells. Here, the operations must maintain the manifold properties of the input data. In a last step, the mixing operation is adapted to the Lie group data. Since we are not in the euclidean space, a mixing operation is not a linear combination of the candidate operations any more because this would violate the $SO_3$ properties.
Following the previous section, a rotation cell is a directed acyclic graph (DAG) composed of an ordered number of nodes and edges, where each node is a latent representation of the Lie group data and each edge corresponds to a valid candidate operation on the rotation group (see Figure 3.2). Each edge is associated with a set of candidate Lie group operations ($\mathcal{O}_R$) that transforms the Lie group valued latent representation from a source node $X_R^{(i)}$ to a target node $X_R^{(j)}$. The intermediate transformation between the nodes in a rotation cell is defined as

$$X_R^{(j)} = \arg\min_{X_R^{(j)}} \sum_{i<j} d_R^2 \left( O_R^{(i,j)}(X_R^{(i)}, X_R^{(j)}) \right), \tag{3.1}$$

where $d_R$ denotes the geodesic distance defined in equation 2.1. This transformation

result corresponds to the unweighted Fréchet mean of the predecessor nodes, so that the mixing operations remain on the rotation group. With the new defined rotation cell and its intermediate transformation we can propose a solution to the Lie group architecture search problem.

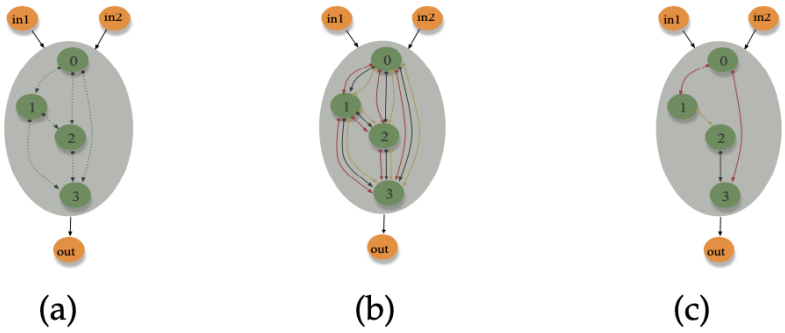


Figure 3.2: (a) A rotation cell composed of 4 intermediate nodes, 2 input and 1 output nod. Initially, the operations on the edges are unknown. (b) Mixture of candidate operations between nodes. (c) Final architecture containing the learned mixing weights. [17]

# Chapter 4

# NAS for Lie group networks

In this chapter we propose a solution to the Lie group architecture search problem, which is a modification of [17] and [10]. First we introduce the search space and afterwards describe the optimization method used to apply the SuperNet method.

## 4.1 Search space

Most of the candidate operations that form the search space are either taken directly from chapter 2 or a combination thereof. We differentiate between two families of operations, namely the normal and reduced ones. The RotMap and BatchNormalize layers are associated to the normal operation family since they keep the output dimensions identical to the input dimensions. The pooling layers, however, are in the family of the reduced operations because they change the dimensions of the feature maps. The operations cannot be combined inter-familiarly because in this case the dimensions do not match. Obviously, the normal operations are used in normal rotation cells and the reduced operations are used in the reduced rotation cells. The candidate operations are defined as follows:

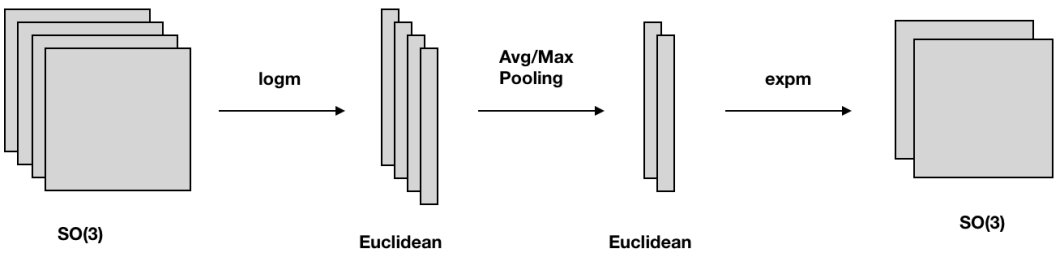


Figure 4.1: Illustration of the Max/Avg Pooling [17]

- **RotMap:** This operation comes in four different flavors, that is, a RotMap layer combined with pre- or post activations, or batch normalization, as listed in table 4.1.
- **Batch normal:** This candidate operation enables us to apply a batch normalization without RotMap layer.
- **Skip normal:** Forwards the input unchanged

- **None normal:** It corresponds to the operation that returns identity matrices as output, i.e., the notion of zero in the $SO_3$ space.

- **MaxPooling:** Corresponds to LieNet's RotPooling layer. As explained in chapter 2, this operation applies a conventional max pooling operation on the axis-angle representation of the rotation matrices. Pooling can be done in spatial or temporal domain.

- **AvgPooling:** Similar to the MaxPooling operation, our average pooling layer (AvgPooling) replaces the max operation by the average operation. Especially in the temporal domain, it can be beneficial to keep the average skeletal features values instead of the maximum ones because the average value represents a sequence of frames better. The AvgPooling layer is defined as

$$\begin{aligned} & f_a^{(k)}\left(\{(\mathbf{R}_{1,2}^{k-1,1}...\mathbf{R}_{M-1,M}^{k-1,1}),...,(\mathbf{R}_{1,2}^{k-1,p}...\mathbf{R}_{M-1,M}^{k-1,p})\}\right) \\ & = \left(avg\left(\{\mathbf{R}_{1,2}^{k-1,1},...,\mathbf{R}_{1,2}^{k-1,p}\}\right),...,avg\left(\{\mathbf{R}_{M-1,M}^{k-1,1},...,\mathbf{R}_{M-1,M}^{k-1,p}\}\right)\right), \end{aligned} \tag{4.1}$$

  where M is the number of body parts in one skeleton, p is the number of video frames, and the function $avg(\cdot)$ is defined as

$$avg\left(\{\mathbf{R}_{m,n}^{k-1,1},...,\mathbf{R}_{m,n}^{k-1,p}\}\right) = expm\left(\frac{1}{p}\sum_{i=1}^{p} logm(\mathbf{R}_{m,n}^{k-1,i})\right). \tag{4.2}$$

  A graphical illustration of max and average pooling can be found in Figure 4.1.

- **Skip reduced:** It divides the input into two smaller parts, applies a Max Pooling on each part, and concatenates the results. Here, the output has a reduced dimension in either the spatial or temporal domain.

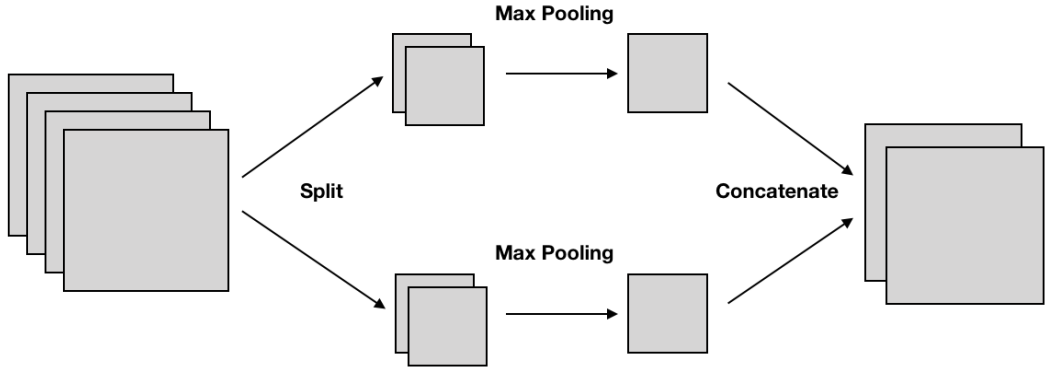


Figure 4.2: Skip reduced operation

| **Operation** | **Definition** |
|---|---|
| RotMap_0 | {RotMap} |
| RotMap_1 | {RotMap, BatchNormalize} |
| RotMap_2 | {RotMap, BatchNormalize, ReLU} |
| RotMap_3 | {ReLU, RotMap, BatchNormalize} |
| Batch_normal | {BatchNormalize} |
| Skip_normal | {Output same as input} |
| None_normal | {Return identity matrix} |
| MaxPooling_spatial | {Spatial RotPooling} |
| MaxPooling_temporal | {Temporal RotPooling} |
| AvgPooling_spatial | {Spatial AvgPooling} |
| AvgPooling_temporal | {Temporal AvgPooling} |
| Skip_reduced_spatial | $C_i$ = MaxPooling_spatial$(R_i); i = 1, 2$, $C_{out} = concat(C_1, C_2)$ |
| Skip_reduced_temporal | $C_i$ = MaxPooling_temporal$(R_i); i = 1, 2$, $C_{out} = concat(C_1, C_2)$ |

Table 4.1: Input search space for the Lie group architecture search

## 4.2 SuperNet search

The optimal Lie group architecture search is done via an optimization over the parametrized SuperNet. It stacks the rotation cells with the parametrized candidate operations from our search space in a one-shot search manner. Since the defined search space is discrete, we relax the categorical choice of a particular operation to a softmax over all possible operation using the weighted Fréchet mean. Formally,

$$O_R(X_R) = \arg\min_{X_R^\mu} \sum_{k=1}^{N_l} \alpha^k d_R^2 \left( O_R^{(k)}(X_R), X_R^\mu \right) = exp_{X_\mu} \left[ (\sum_{k=1}^{N_l} \alpha^k)^{-1} \sum_{k=1}^{N_l} \alpha^k log_{X_\mu}(O_R^{(k)}(X_R)) \right] \tag{4.3}$$

where $O_R^{(k)}$ is the $k^{th}$ candidate operation between nodes, $X_\mu$ is the intermediate Fréchet mean from equation 2.2.4, and $N_l$ denotes the number of edges. As usual, the Fréchet mean is computed using the Karcher flow algorithm. The weights are fed through a softmax such that they sum up to 1. An example of a simplified mixed operation is illustrated in Figure 4.3. We consider a cell consisting of three nodes, two input nodes (1 and 2) and one output nodes (node 3). A mixed operation is computed on each input node, where a candidate operation corresponds to an edge. Note that in this case the search space contains only two candidate operations. Each candidate operation has a weight $\alpha_{i\_j}$, where i corresponds to the node index and j to the operation index. In this example, it holds i, j $\in 1, 2$, and $\boldsymbol{\alpha_1} = \{\alpha_{1\_1}, \alpha_{1\_2}\}, \boldsymbol{\alpha_2} = \{\alpha_{2\_1}, \alpha_{2\_2}\}$. Different from [17], we first compute the outputs of of the mixed operations applied to node 1 and 2 separately. Afterwards, we fuse the outputs by computing another Fréchet mean. In this way, the dimensionality remains the same and the data is guaranteed to stay on the

Lie group.

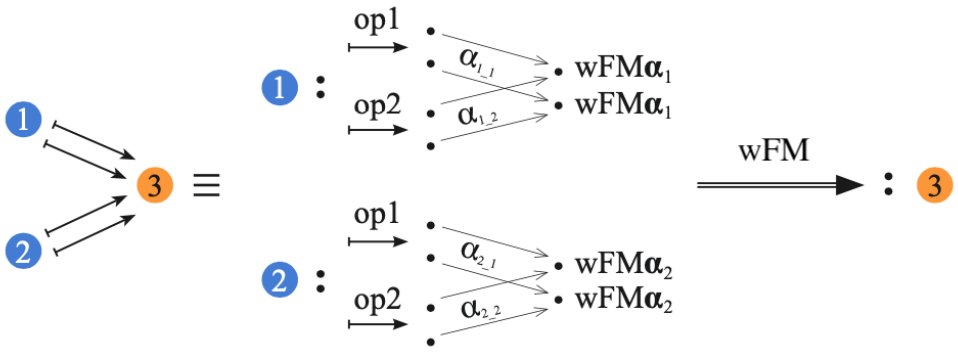


Figure 4.3: Illustration of a simplified mixed operation with two input nodes and one output node. A mixed operation is applied to each input node separately and the outputs are combined using the Fréchet mean in order to keep the dimensionality identical to the input.

Using equation 4.3, the architecture search is done by learning a set of variables $\alpha = \{\alpha^k, \forall k \in N_l\}$. We simultaneously learn the contribution of several possible operation within all the mixed operations ($w$) and the corresponding architecture $\alpha$. That means, for a given $w$ we optimize for $\alpha$ and vice-versa, resulting in a bi-level optimization problem. The lower-level problem corresponds to the optimal weight parameters learned for a given architecture $\alpha$, i.e., $w^{opt}(\alpha)$ using a training loss ($\mathbf{E}^L_{train}(w, \alpha)$. The upper-level optimization updates the variable $\alpha$ given the optimal $w$ using a validation loss ($\mathbf{E}^U_{val}$). The bi-level optimization problem is defined as

$$\underset{\alpha}{\text{minimize}}\ \mathbf{E}^U_{val}\left(w^{opt}(\alpha), \alpha\right); \text{subject to: } w^{opt}(\alpha) = \arg\min_{w} \mathbf{E}^L_{train}(w, \alpha) \tag{4.4}$$

This approach leads to an optimal mixture of architecture. At the end of the optimization, the discrete architecture is obtained by replacing each mixed operation $O^{(k)}_R$ with the most likely operation, i.e., $o^{(k)} = \arg\max_{o^{(k)} \in \mathcal{O}} \alpha^{(k)}_o$.

**Bi-level optimization:** Solving this optimization problem is difficult because the inner optimization is computationally very expensive. For this reason, we approximate equation 4.4 as follows:

$$\nabla_\alpha \mathbf{E}^U_{val}\left(w^{opt}(\alpha), \alpha\right) \approx \nabla_\alpha \mathbf{E}^U_{val}\left(w - \eta \nabla_w \mathbf{E}^L_{train}(w, \alpha), \alpha\right) \tag{4.5}$$

Here, $\eta$ is the learning rate and $\nabla$ represents the gradient operator. In this case, the gradients computation is adapted to the Lie group $SO_3$, following [6]. Applying the chain rule to equation 4.5 yields

$$\nabla_\alpha \mathbf{E}^U_{val}(\tilde{w}, \alpha) - \eta \nabla^2_{\alpha,w} \mathbf{E}^L_{train}(w, \alpha) \nabla_{\tilde{w}} \mathbf{E}^U_{val}(\tilde{w}, \alpha), \tag{4.6}$$

where $\tilde{w} = \Psi_r\left(w - \eta \tilde{\nabla}_w \mathbf{E}^L_{train}(w, \alpha\right)$ denotes the weight update on the $SO_3$ manifold for

the forward model. $\tilde{\nabla}_w$ and $\Psi_r$ are the Riemannian gradient and the retraction operator, respectively. The second term in equation 4.6 contains second order differentials with high computationally complexity, therefore, using the finite approximation method, the second term of equation 4.6 reduces to

$$\nabla^2_{\alpha,w}\mathbf{E}^L_{train}(w,\alpha)\nabla_{\tilde{w}}\mathbf{E}^U_{val}(\tilde{w},\alpha) = \left(\nabla_\alpha \mathbf{E}^L_{train}(w^+,\alpha) - \nabla_\alpha \mathbf{E}^L_{train}(w^-,\alpha)\right)/2\delta, \tag{4.7}$$

where $w^\pm = \Psi_r\left(w \pm \delta\tilde{\nabla}_{\tilde{w}}\mathbf{E}^U_{val}(\tilde{w},\alpha)\right)$ and $\delta$ is a small number set to $0.01/\parallel \nabla_{\tilde{w}}\mathbf{E}^U_{val}(\tilde{w},\alpha) \parallel_2$ [17] [10].
The pseudo code [17] of the bi-level optimization is shown in algorithm 5.

**Algorithm 5** The proposed neural architecture search for adapted to Lie group data [17]

**Require:** Mixed operation $O_R$ which is parametrized by $\alpha^k$ for each edge $k \in N_l$;
**while** not converged **do**
**Step 1:** Update architecture $\alpha$ using equation 4.5.
**Step 2:** Update w by solving $\nabla_w \mathbf{E}_{train}(w,\alpha)$
**end while**
Derive the final architecture based on the learned $\alpha$. The operation at edge k is chosen by $\arg\max_{o\in O_R}\{\alpha^k_o\}$

# Chapter 5

# Experimental results

Our new models LieNetBN and LieNetNAS are evaluated on the gaming action dataset G3D. This dataset contains 10 subjects performing 20 gaming actions: punch right, punch left, kick right, kick left, defend, golf swing, tennis swing forehand, tennis swing backhand, tennis serve, throw bowling ball, aim and fire gun, walk, run, jump, climb, crouch, steer a car, wave, flap and clap [3]. Here, one sample consists of a video sequence of 100 frames and each frame contains 342 rotation matrices, yielding an object of dimension (342, 100, 3, 3).

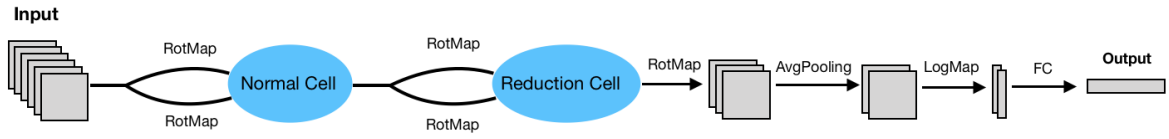


Figure 5.1: Initialization of the LieNetNAS architecture

**LieNetNAS:** As shown in figure 5.1, our configuration consists of a normal cell, a reduction cell, a RotMap layer and a AvgPooling layer, in this order. At the end, analogous to the LieNet model, we use a LogMap to flatten the data, pass it to the fully connected layer and feed it into the softmax classifier. In our framework, we use 4 nodes within a cell which includes two input nodes, one intermediate node and one output node. The inputs to a cell are preprocessed with a RotMap layer. The dataset contains 666 samples in total and we assign 333 samples to each the training and test set, respectively. We train the network with a batch size of 16 in order to not exceed the memory allocation limits. Furthermore, we use momentum SGD to optimize the weights $w$, with initial learning rate $\eta_w = 0.025$ (annealed down to zero following a cosine schedule without restart [11]), momentum 0.9, and weight decay $3 \times 10^{-4}$, as in [10]. For both rotation cells, we initialize the architecture variables equally, i.e., the $\alpha$'s are distributed uniformly such that they all have the same amount of importance. We use the Adam optimizer [8] for $\alpha$, with initial learning rate $\eta_\alpha = 3 \times 10^{-4}$, momentum $\beta = 0.9$ for both cells, and weight decay $10^{-3}$. Training takes 6 CPU-hours for a total of 50 epochs. There was no computational speedup when using a GPU. As visible in figure 5.2, the accuracies increase very fast at the beginning because the learning rate for the weights $w$ is rather high. As the number of epochs augments, the learning rate diminishes and the accuracies stabilize. The abrupt

| Method | G3D-Gaming |
|---|---|
| RBM+HMM [13] | 86.40% |
| SE [14] | 87.23% |
| SO [20] | 87.95% |
| LieNet-0Block | 84.55% |
| LieNet-1Block | 85.16% |
| LieNet-2Blocks | 86.67% |
| LieNet-3Blocks | 89.10% |
| **LieNetBN** | **90.4 %** |
| **LieNetNAS** | **90.9 %** |

Table 5.1: Test accuracies on the G3D-Gaming dataset.

changes in the accuracy curves between two epochs occur due to the architecture changing in the corresponding epoch, i.e., an updated architecture can generalize much better or worse than the previous one and hence the accuracy will rise or drop correspondingly.

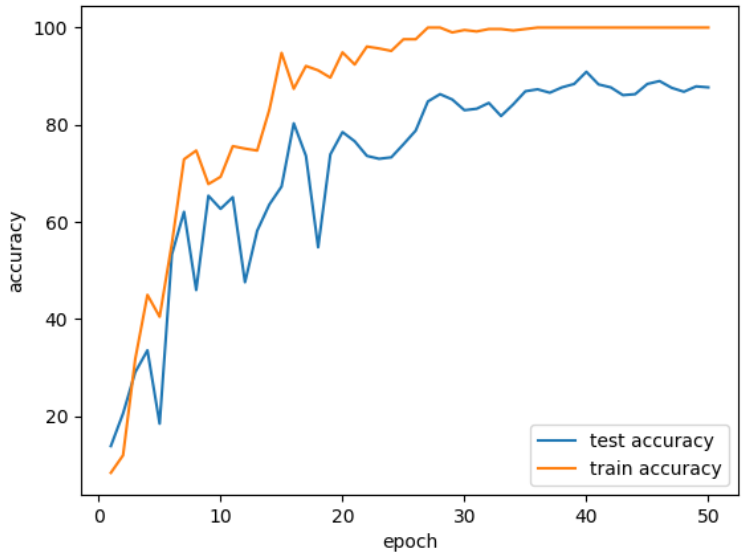


Figure 5.2: Evolution of train and test accuracy for LieNetNAS

After running the architecture search, we trained the resulting model from scratch. Hereby, following [6], the learning rate is fixed to $\lambda = 0.01$ without momentum and the batch size is again 16. In this case, there was no further increase of the test accuracy. Table 5.1 shows the recognition accuracies of several models. LieNetNAS has a slightly higher accuracy than LieNet-3Blocks and LieNetBN.
As mentioned, we follow a cross-subject test setting, where half the subjects are used for training and the other half are employed for testing [6]. All the results reported for this dataset are averaged over 20 different combinations of training and testing datasets.
The learned reduction cell and normal cell are shown in figures 5.3 and 5.4, respectively.
**LieNetBN:** Our second model is depicted in figure 5.5. Similar to LieNet-3Blocks, it is divided into three blocks, but in the second and third blocks we use the AvgPooling layer

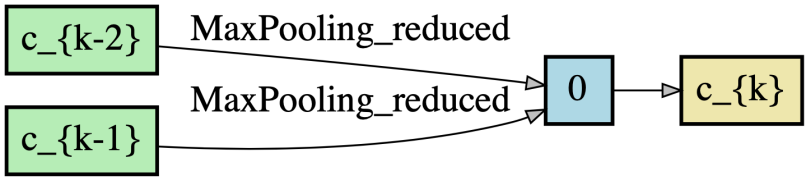


Figure 5.3: Learned reduction cell

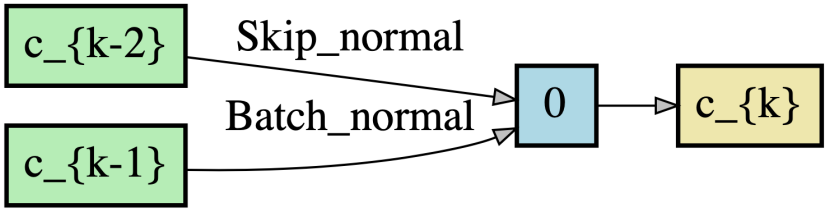


Figure 5.4: Learned normal cell

and we also apply a batch normalization after the third RotPooling layer. At the end, the data is flattened, passed to the fully connected layer and fed into the softmax classifier. This configuration is trained with a learning rate fixed to $\lambda = 0.01$, no momentum and batch size 128. The introduction of the batch normalization in LieNetNAS and LieNetBN

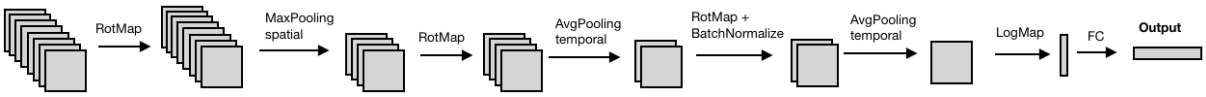


Figure 5.5: LieNetBN architecture

leads to a re-distribution of the input data and hence to an improvement in accuracy. Figure 5.6 shows an example skeleton sequence with label "tennis swing forehand". While LieNet-3Blocks misclassified this sample as "punch right", our new models classified it correctly.

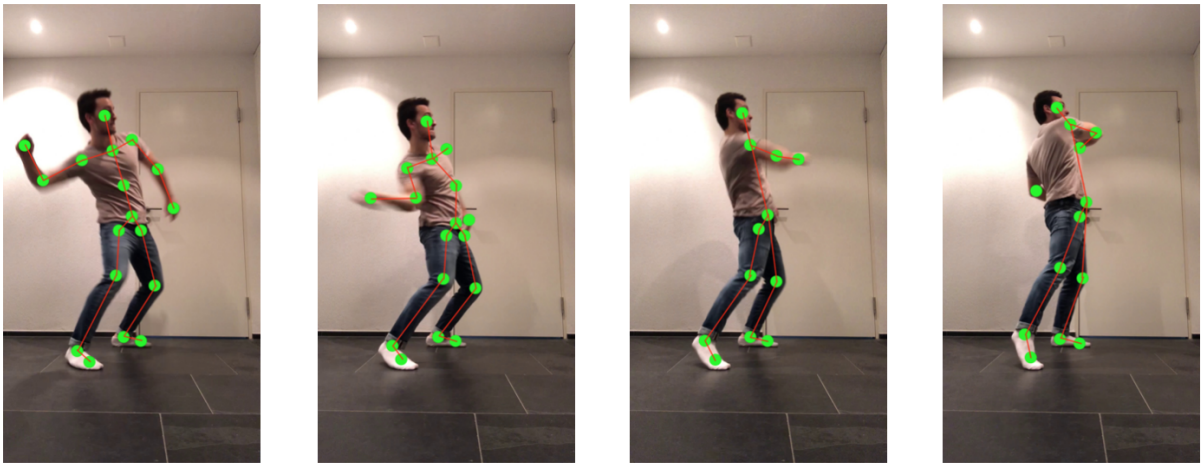

Figure 5.6: Training sample with label "tennis swing forehand". This example was misclassified by LieNet-3Blocks, but classified correctly by LieNetBN and LieNetNAS.

# Chapter 6

# Conclusion & future work

The goal of this project is to develop a new human action recognition network using neural architecture search and improve the existing model LieNet and its extension LieNetBN. This objective has been fulfilled as the NAS model has a slightly higher accuracy than the hand designed ones. Architecture search is realized by introducing computation cells and a set of candidate operations which handle Lie group data such as rotation matrices. Making use of the continuous relaxation of the search space, the architecture and model weights can be optimized simultaneously using gradient descent. The new framework is evaluated on the G3D Gaming dataset.
The main issue during training is the high memory allocation due to the fact that the newly introduced LieNet layers are computationally expensive and need to be further optimized. In addition, this restricts us to only stack two cells at most. One approach to solve this issue is to split the training into multiple rounds, i.e., after each round the current parameters are stored and afterwards reloaded in the next round. In this way we avoid training interruption due to excessive memory allocation. The problem hereby is that the learning rate scheduler starts from scratch in each training round, meaning that the learning rate is not optimized as good as if we would train without interruption. Some proposals for further improvement of the neural architecture search are:

- To optimize the LieNet layers in terms of time complexity and resource allocation
- To enlarge the current search space by non-linear operations
- To include more nodes within a computation cell
- To extend the architecture search to other data sets

# Bibliography


[1] Rushil Anirudh, Pavan Turaga, Jingyong Su, and Anuj Srivastava. Elastic functional coding of riemannian trajectories. *IEEE transactions on pattern analysis and machine intelligence*, 39(5):922–936, 2016.

[2] Rajendra Bhatia and John Holbrook. Riemannian geometry and matrix geometric means. *Linear algebra and its applications, 413(2-3):594-618*, 2006.

[3] V. Bloom, D. Makris, and V. Argyriou. G3d: A gaming action dataset and real time action recognition evaluation framework. *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, page 9, 2012.

[4] Daniel Brooks, Olivier Schwander, Frederic Barbaresco, Jean-Yves Schneider, and Matthieu Cord. Riemannian batch normalization for spd neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 15489–15500. Curran Associates, Inc., 2019.

[5] Rudrasis Chakraborty. Manifoldnorm: Extending normalizations on riemannian manifolds. *arXiv preprint arXiv:2003.13869*, 2020.

[6] Zhiwu Huang, Chengde Wan, Thomas Probst, and Luc Van Gool. Deep learning on lie groups for skeleton-based action recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[7] Hermann Karcher. Riemannian center of mass and mollifier smoothing. *Communications on pure and applied mathematics, 30(5):509-541*, 1977.

[8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search, 2018.

[10] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[11] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[12] Maher Moakher. A differential geometric approach to the geometric mean of symmetric positive-definite matrices. *SIAM Journal on Matrix Analysis and Applications, 26(3):735-747*, 2005.

[13] S. Nie and Q. Ji. Capturing global and local dynamics for human action recognition. *In ICPR*, 2014.

[14] F. Arrate R. Vemulapalli and R. Chellappa. Human action recognition by representing 3d skeletons as points in a lie group. *In ICPR*, 2014.

[15] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search, 2019.

[16] Attaullah Sahito, M Abdul Rahman, and Jamil Ahmed. Adaptive error detection method for p300-based spelling using riemannian geometry. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, 7(11):332–337, 2016.

[17] Rhea Sukthanker. Neural architecture search of spd manifold nets and its end to end extension. CVL Lab, ETH Zurich, 2020.

[18] Rhea Sanjay Sukthanker, Zhiwu Huang, Suryansh Kumar, Erik Goron Endsjo, Yan Wu, and Luc Van Gool. Neural architecture search of spd manifold networks, 2020.

[19] Yuan Tian, Qin Wang, Zhiwu Huang, Wen Li, Dengxin Dai, Minghao Yang, Jun Wang, and Olga Fink. Off-policy reinforcement learning for efficient and effective gan architecture search. In *European Conference on Computer Vision*, pages 175–192. Springer, 2020.

[20] R. Vemulapalli and R. Chellappa. Rolling rotations for recognizing human actions from 3d skeletal data. *In CVPR*, 2016.

[21] Raviteja Vemulapalli, Felipe Arrate, and Rama Chellappa. Human action recognition by representing 3d skeletons as points in a lie group. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 588–595, 2014.

[22] Raviteja Vemulapalli and Rama Chellapa. Rolling rotations for recognizing human actions from 3d skeletal data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4471–4479, 2016.

[23] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2017.

[24] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.